IUP

Portable User Interface

Version 3.17

IUP is a multi-platform toolkit for building graphical user interfaces. It offers a simple API in three basic languages: C, Lua and LED. **IUP**'s purpose is to allow a program source code to be compiled in different systems without any modification. Its main advantages are:

- high performance, due to the fact that it uses native interface elements.
- fast learning by the user, due to the simplicity of its API.

This work was developed at Tecgraf/PUC-Rio by means of the partnership with PETROBRAS/CENPES.

Project Management:

Antonio Escaño Scuri

Tecgraf - Computer Graphics Technology Group, PUC-Rio, Brazil
http://www.tecgraf.puc-rio.br/iup
Also available at http://iup.sourceforge.net/

SOURCEFORGE.NET

Veja esta página em Português.

Product

Overview

IUP is a multi-platform toolkit for building graphical user interfaces. It offers APIs in three basic languages: C, Lua and LED.

Its library contains about 100 functions for creating and manipulating dialogs.

IUP's purpose is to allow a program to run in different systems without changes - the toolkit provides the application portability. Supported systems include: GTK+, Motif and Windows.

IUP uses an abstract layout model based on the boxes-and-glue paradigm from the TEX text editor. This model, combined with the dialog-specification language (LED) or with the Lua binding (IupLua) makes the dialog creation task more flexible and independent from the graphics system's resolution.

Currently available interface elements can be categorized as follows:

- **Primitives** (effective user interaction): **dialog, label, button, text, multi-line, list, toggle, canvas, frame, image**.
- **Composition** (ways to show the elements): **hbox, vbox, zbox, fill**.
- **Grouping** (definition of a common functionality for a group of elements): **radio**.
- **Menu** (related both to menu bars and to pop-up menus): **menu, submenu, item, separator**.
- Additional (elements built outside the main library): **dial, gauge, matrix, tabs, valuator, OpenGL canvas, color chooser, color browser**.
- **Dialogs** (useful predefined dialogs): **file selection, message, alarm, data input, list selection**.

Hence IUP has some advantages over other interface toolkits available:

- **Simplicity:** due to the small number of functions and to its attribute mechanism, the learning curve for a new user is often faster.
- **Portability:** the same functions are implemented in each one of the platforms, thus assuring the interface system's portability.
- **Customization:** the dialog specification language (LED) and the Lua binding (IupLua) are two mechanisms in which it is possible to customize an application for a specific user with a simple-syntax text file.
- **Flexibility**: its abstract layout mechanism provides flexibility to dialog creation.
- **Extensibility:** the programmer can create new interface elements as needed.

IUP is free software, can be used for public and commercial applications.

Availability

The library is available for several **compilers**:

- GCC and CC, in the UNIX environment
- Visual C++, Borland C++, Watcom C++ and GCC (Cygwin and MingW), in the Windows environment

The library is available for several **operating systems**:

- UNIX (SunOS, IRIX, and AIX) using Motif 2.x
- UNIX (FreeBSD and Linux) using GTK+ (since 3.0)
- Microsoft Windows XP/2003/Vista/7 using the Win32 API

Support

The official support mechanism is by e-mail, using **iup@tecgraf.puc-rio.br**. Before sending your message:

- Check if the reported behavior is not described in the user guide.
- Check if the reported behavior is not described in the specific control or driver characteristics.
- Check the History to see if your version is updated.
- Check the To Do list to see if your problem has already been reported.

If all these points were checked, you can report your problem. Please specify in your message: **function, attribute, callback, platform** and **compiler**.

We host the **IUP** support features at **SourceForge**: http://sourceforge.net/projects/iup/. It provides us Mailing List, SVN Repository and Downloads.

The discussion list is available at: http://lists.sourceforge.net/lists/listinfo/iup-users.
Source code, pre-compiled binaries and documentation can be downloaded at: http://sourceforge.net/projects/iup/files/.
The SVN can be browsed at: https://sourceforge.net/p/iup/iup/.

If you want us to develop a specific feature for the toolkit, Tecgraf is available for partnerships and cooperation.

Lua documentation and resources can be found at http://www.lua.org/.

Credits

This work was developed at Tecgraf by means of the partnership with PETROBRAS/CENPES.

Library Authors:

- Marcelo Gattass

- Luiz Henrique de Figueiredo
- Carlos Henrique Levy
- Antonio Scuri

We must also mention engineer Enio Emanuel Russo, from PETROBRAS, who effectively contributed to the system's specification and project.

Thanks to the people that worked and contributed to the library:

- AndrÃ© Carregal
- AndrÃ© Clinio
- AndrÃ© Costa
- AndrÃ© Derraik
- Carlos Augusto Mendes
- Carlos JosÃ© Pereira de Lucena
- Claudio Coutinho de Biasi
- Danny Reinhold
- Diego Nehab
- Diogo Martinez
- Guilherme Fonseca Alvarenga
- Henrique Dalcin Mendes Pinheiro
- Heesob Park
- Leonardo Constantino Oliveira
- Luiz CristÃ³vÃ£o Gomes Coelho
- Luiz Martins
- Marian Trifon
- Mark Stroetzel Glasberg
- Mauricio Oliveira Carneiro
- Milton Jonathan
- Neil Armstrong Rezende
- Nicolas Noble
- Otfried Cheong
- Rafael Rieder
- Renato Borges
- Renato Cerqueira
- Roberto Beauclair
- Steve Donovan
- Tomas Guisasola Gorham
- Vinicius Almendra
- Warren Music

Thanks for the SourceForge for hosting the support features. Thanks for the LuaForge team for previously hosting the support features for many years.

IUP is registered at the National Institute of Intellectual Property in Brazil (INPI) under the number 07569-0, and so it is protected against illegal use. See the Tecgraf Library License for further usage information and Copyright.

Documentation

This documentation is available at http://www.tecgraf.puc-rio.br/iup and http://iup.sourceforge.net/

The full documentation can be downloaded from the Download Files. The documentation is also available in Adobe Acrobat and Windows HTML Help formats.

The HTML navigation uses the WebBook tool, available at http://www.tecgraf.puc-rio.br/webbook.

There are also a few presentations:

- Lua Workshop 2009 - IUP, CD and IM in Lua (http://www.lua.org/wshop09.html#Scuri) [iupcdim_wlua2009.pdf]
- PUCRS 2010 - IUP, CD and IM [iupcdim_facin2010.pdf]

Publications

This product stimulated the following scientific publications:

- Scuri, A. "IUP - Portable User Interface". Software Developer's Journal. Dec/2005. [iup_sdj2005.pdf]
- Levy, C. H.; Figueiredo, L. H.; Gattass, M.; Lucena, C.; and Cowan, D. "IUP/LED: A Portable User Interface Development Tool". *Software: Practice & Experience*, 26 #7 (1996) 737-762. [spe95.pdf]
- Oliveira Prates, R.; Figueiredo, L. H.; and Gattass, M. "EspecificaÃ§Ã£o de Layout Abstrato por ManipulÃ§Ã£o Direta". Proceedings of VII SIBGRAPI (1994), 165-172. [sib94.pdf in Portuguese]
- Oliveira Prates, R.; Gattass, M. ;and Figueiredo, L. H. "Visual LED: uma ferramenta interativa para geraÃ§Ã£o de interfaces grÃificas". M.Sc. dissertation, Computer Science Department, PUC-Rio, 1994. [prates94.pdf in Portuguese]
- Levy, C. H. "IUP/LED: Uma Ferramenta PortÃitil de Interface com UsuÃirio". M.Sc. dissertation, Computer Science Department, PUC-Rio, 1993.[levy93.pdf in Portuguese]
- Figueiredo, L. H.;Gattass, M.; and Levy, C.H. "Uma EstratÃ©gia de Portabilidade para AplicaÃ§Ãµes GrÃificas Interativas". Proceedings of VI SIBGRAPI (1993), 203-211. [sib93.pdf in Portuguese]

Interview at the FLOSS weekly show about Free Libre Open Source Software, hosted by Randal Schwartz:

- http://twit.tv/show/floss-weekly/190

Tecgraf Library License

The Tecgraf products under this license are: IUP, CD and IM.

All the products under this license are free software: they can be used for both academic and commercial purposes at absolutely no cost. There are no paperwork, no royalties, no GNU-like "copyleft" restrictions, either. Just download and use it. They are licensed under the terms of the MIT license reproduced below, and so are compatible with GPL and also qualifies as Open Source software. They are not in the public domain, PUC-Rio keeps their copyright. The legal details are below.

The spirit of this license is that you are free to use the libraries for any purpose at no cost without having to ask us. The only requirement is that if you do use them, then you should give us credit by including the copyright notice below somewhere in your product or its documentation. A nice, but optional, way to give us further credit is to include a Tecgraf logo and a link to our site in a web page for your product.

The libraries are designed, implemented and maintained by a team at Tecgraf/PUC-Rio in Brazil. The implementation is not derived from licensed software. The library was developed by request of Petrobras. Petrobras permits Tecgraf to distribute the library under the conditions here presented.

Download

The download site for pre-compiled binaries, documentation and sources is at **SourceForge**:

http://sourceforge.net/projects/iup/files/

Use this link for the latest version: http://sourceforge.net/projects/iup/files/3.17/

Before downloading any precompiled binaries, you should read before the Tecgraf Library Download Tips.

Some other files are available directly at the **IUP** download folder:

http://www.tecgraf.puc-rio.br/iup/download/

Tecgraf/PUC-Rio Library Download Tips

All the libraries were build using **Tecmake**. Please use it if you intend to recompile the sources. **Tecmake** can be found at http://www.tecgraf.puc-rio.br/tecmake.

The **IM** files can be downloaded at http://sourceforge.net/projects/imtoolkit/files/.
The **CD** files can be downloaded at http://sourceforge.net/projects/canvasdraw/files/.
The **IUP** files can be downloaded at http://sourceforge.net/projects/iup/files/.
The **Lua** files can be downloaded at http://sourceforge.net/projects/luabinaries/files/.

Build Configuration

Libraries and executables were built using speed optimization. In UNIX the dynamic libraries were NOT built with the -fpic parameter. In MacOS X the dynamic libraries are in bundle format. The source code along with the "config.mak" files for **Tecmake** are also available.

The DLLs were built using the **cdecl** calling convention. This should be a problem for Visual Basic users.

In Visual C++ 6 and 7 we use the single thread C Run Time Library for static libraries and the multi thread C RTL for DLLs. Because this were the default in Visual Studio for new projects. Since Visual C++ 8, both use the multithread C RTL.

Packaging

The package files available for download are named according to the platform where they were build.

In UNIX all strings are based in the result of the command "uname -a". The package name is a concatenation of the platform **uname**, the system **major** version number and the system **minor** version number. Some times a suffix must be added to complement the name. The compiler used is always gcc. Binaries for 64-bits receive the suffix: "_64". In Linux when there are different versions of gcc for the same uname, the platform name is created adding the major version number of the compiler added as a suffix: "g3" for gcc 3 and "g4" for gcc 4.

In Windows the platform name is the **compiler** and its **major** version number.

All library packages **(*_lib*)** contains pre-compiled binaries for the specified platform and includes. Packages with **"_bin"** suffix contains executables only.

The package name is a general reference for the platform. If you have the same platform it will work fine, but it may also work in similar platforms.

Here are some examples of packages:

iup2_4_Linux26_lib.tar.gz = IUP 2.4 32-bits Libraries and Includes for Linux with Kernel version 2.6 built with gcc 3.
iup2_4_Linux26g4_64_bin.tar.gz = IUP 2.4 64-bits Executables for Linux with Kernel version 2.6 built with gcc 4.
iup2_4_Win32_vc8_lib.tar.gz = IUP 2.4 32-bits Static Libraries and Includes for Windows to use with Visual C++ 8 (2005).
iup2_4_Win32_dll9_lib.tar.gz = IUP 2.4 32-bits Dynamic Libraries (DLLs), import libraries and Includes for Windows to use with Visual C++ 9 (2008).
iup2_4_Docs_html.tar.gz = IUP 2.4 documentation files in HTML format (the web site files can be browsed locally).
iup2_4_Win32_bin.tar.gz = IUP 2.4 32-bits Executables for Windows.

The documentation files are in HTML format. They do not include the CHM and PDF versions. These two files are provided as a separate download, but they all have the same documentation.

Installation

For any platform we recommend you to create a folder to contain the third party libraries you download. Then just unpack the packages you download in that folder. The packages already contains a directory structure that separates each library or toolkit. For example:

```
\mylibs\
        iup\
            bin\
            html\
            include\
            lib\Linux26
            lib\Linux26g4_64
            lib\vc8
            src
        cd\
        im\
        lua5.1\
        lua52\
```

This structure will also made the process of building from sources more simple, since the projects and makefiles will assume this structure .

Usage

For makefiles use:

```
1) "-I/mylibs/iup/include" to find include files
2) "-L/mylibs/iup/lib/Linux26" to find library files
3) "-liup" to specify the library files
```

For IDEs the configuration involves the same 3 steps above, but each IDE has a different dialog. The IUP toolkit has a Guide for some IDEs:

Borland C++ BuilderX - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/cppbx.html
Code Blocks - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/codeblocks.html
Dev-C++ - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/dev-cpp.html
Eclipse for C++ - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/eclipse.html
Microsoft Visual C++ (Visual Studio 2003) - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/msvc.html
Microsoft Visual C++ (Visual Studio 2005) - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/msvc8.html
Open Watcom - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/owc.html

Available Platforms

The following platforms can be available:

Package Name	Description

AIX43	IBM AIX 4.3 (ppc) / gcc 2.95 (Motif 2.1)
IRIX65	SGI IRIX 6.5 (mips) / gcc 3.0 (Motif 2.1)
IRIX6465	SGI IRIX 6.5 (mips) / gcc 3.3 (Motif 1.2)
Linux26g4	Ubuntu 10.4 (x86) / Kernel 2.6 / gcc 4.4 (GTK 2.20)
Linux26g4_64	Ubuntu 10.4 (x64) / Kernel 2.6 / gcc 4.4 (GTK 2.20)
Linux30	Ubuntu 11.10 (x86) / Kernel 3.0 / gcc 4.6 (GTK 2.24)
Linux30_64	Ubuntu 11.10 (x64) / Kernel 3.0 / gcc 4.6 (GTK 2.24)
Linux32	Ubuntu 12.04 (x86) / Kernel 3.2 / gcc 4.6 (GTK 2.24)
Linux32_64	Ubuntu 12.04 (x64) / Kernel 3.2 / gcc 4.6 (GTK 2.24)
Linux35_64	Ubuntu 12.10 (x64) / Kernel 3.5 / gcc 4.7 (GTK 2.24)
Linux313_64	Ubuntu 14.04 (x64) / Kernel 3.13 / gcc 4.8 (GTK 3.10)
Linux319_64	Ubuntu 15.04 (x64) / Kernel 3.19 / gcc 4.9 (GTK 3.14)
SunOS510	Sun Solaris 10 (sparc) / gcc 3.4 (Motif 2.1)
SunOS510x86	Sun Solaris 10 (x86) / gcc 3.4 (Motif 2.1)
FreeBSD54	Free BSD 5.4 (x86) / gcc 3.4
MacOS104	Mac OS X 10.4 (ppc) [Tiger] / Darwin Kernel 8 / gcc 4.0
MacOS104x86	Mac OS X 10.4 (x86) [Tiger] / Darwin Kernel 8 / gcc 4.0
MacOS105x86	Mac OS X 10.5 (x86) [Leopard] / Darwin Kernel 9 / gcc 4.0
MacOS106	Mac OS X 10.6 (x64) [Snow Leopard] / Darwin Kernel 10 / gcc 4.2
MacOS107	Mac OS X 10.7 (x64) [Lion] / Darwin Kernel 11 / gcc 4.2
MacOS109	Mac OS X 10.9 (x64) [Mavericks] / Darwin Kernel 13 / gcc 4.2
Win32_vc8	Static library built with Microsoft Visual C++ 8.0 (2005) (static RTL/multithread) Also compatible with Microsoft Visual C++ 2005 Express Edition
Win32_vc9	Static library built with Microsoft Visual C++ 9.0 (2008) (static RTL/multithread) Also compatible with Microsoft Visual C++ 2008 Express Edition
Win32_vc10	Static library built with Microsoft Visual C++ 10.0 (2010) (static RTL/multithread) Also compatible with Microsoft Visual C++ 2010 Express Edition
Win32_vc11	Static library built with Microsoft Visual C++ 11.0 (2012) (static RTL/multithread) Also compatible with Microsoft Visual C++ 2012 Express Edition
Win32_vc12	Static library built with Microsoft Visual C++ 12.0 (2013) (static RTL/multithread) Also compatible with Microsoft Visual C++ 2013 Express Edition - http://www.microsoft.com/express/vc/ Â¹
Win32_vc14	Static library built with Microsoft Visual C++ 14.0 (2015) (static RTL/multithread) Also compatible with Microsoft Visual C++ 2015 Express Edition - http://www.microsoft.com/express/vc/ Â¹
Win32_dll8	DLL and import library built with vc8, creates dependency with MSVCR80.DLL
Win32_dll9	DLL and import library built with vc9, creates dependency with MSVCR90.DLL
Win32_dll10	DLL and import library built with vc10, creates dependency with MSVCR100.DLL
Win32_dll11	DLL and import library built with vc11, creates dependency with MSVCR110.DLL
Win32_dll12	DLL and import library built with vc12, creates dependency with MSVCR120.DLL
Win32_dll14	DLL and import library built with vc14, creates dependency with VCRUNTIME140.DLL
Win64_vc8	Same as **Win32_vc8** but for 64-bits systems using the x64 standard.
Win64_vc9	Same as **Win32_vc9** but for 64-bits systems using the x64 standard.
Win64_vc10	Same as **Win32_vc10** but for 64-bits systems using the x64 standard.
Win64_vc11	Same as **Win32_vc11** but for 64-bits systems using the x64 standard.
Win64_vc12	Same as **Win32_vc12** but for 64-bits systems using the x64 standard.
Win64_dll8	Same as **Win32_dll8** but for 64-bits systems using the x64 standard.
Win64_dll9	Same as **Win32_dll9** but for 64-bits systems using the x64 standard.
Win64_dll10	Same as **Win32_dll10** but for 64-bits systems using the x64 standard.
Win64_dll11	Same as **Win32_dll11** but for 64-bits systems using the x64 standard.
Win64_dll12	Same as **Win32_dll12** but for 64-bits systems using the x64 standard.
Win64_dll14	Same as **Win32_dll14** but for 64-bits systems using the x64 standard.
Win32_gcc4	Static library built with Cygwin gcc 4.3 (Depends on Cygwin DLL 1.7) - http://www.cygwin.com/ Â¹
Win32_cygw17	Same as **Win32_gcc4**, but using the Cygwin Posix system and also with a DLL and import library
Win32_dllg4	DLL and import library built with Cygwin gcc 4.3 (See **Win32_gcc4**)
Win32_mingw4	Static library built with MingW gcc 4.6 - http://www.mingw.org/ Â¹ Also compatible with Dev-C++ - http://www.bloodshed.net/devcpp.html and with Code Blocks - http://www.codeblocks.org/ Â¹
Win32_dllw4	DLL and import library built with MingW gcc 4.6 (See **Win32_mingw4**)
Win64_mingw4	Static library built with MingW gcc 4.5 - http://mingw-w64.sourceforge.net/ Â¹ Tool chains targeting Win64 / Personal Builds / "sezero" for 64-bits systems using the x64 standard.
Win64_dllw4	DLL and import library built with MingW gcc 4.5, for 64-bits systems using the x64 standard. creates dependency with MSVCRT.DLL
Win32_owc1	Static library built with Open Watcom 1.5 - http://www.openwatcom.org/
Win32_bc55	Static library built with Borland C++ 5.5 Compiler - https://downloads.embarcadero.com/free/c_builder Â¹
Win32_bc6	Static library built with Embarcadero C++ Builder 2010 / Embarcadero C++ 6 Compiler - https://downloads.embarcadero.com/free/c_builder (trial)
Win32_bin	Executables only for Windows NT/2000/XP/Vista/7 (can be generated by any of the above compilers)
Win64_bin	Same as **Win32_bin** but for 64-bits systems using the x64 standard

Win32_cygw17_bin	Executables only for Windows NT/2000/XP, but using the Cygwin Posix system (See **Win32_cygw17**)

Â¹ - Notice that all the Windows compilers with links here are free to download and use.
Â² - Recently Borland removed the C++ Builder X from download. But if you bought a book that has the CD of the compiler, then it is still free to use.
[3] - Open Motif 2.2 is classified as 'experimental' by the Open Group.

SVN

The SVN repository is at **SourceForge**. It can also be interactively browsed at:

https://sourceforge.net/p/iup/iup/

To checkout using the command line use:

```
svn checkout svn://svn.code.sf.net/p/iup/iup/trunk/iup iup
```

History of Changes

Version 3.x

See Version 3.x History.

Version 2.x

See Version 2.x History.

Version 1.x

See Version 1.x History.

History of Changes in Version 3.x

Check the Migration Guide for a summary of the important changes and how to proceed when migrating from version 2.x to version 3.x.

Version 3.17 (30/Nov/2015)

- New: LASTSORTCOLUMN attribute for **IupMatrixEx**.
- New: EXEFILENAME global attribute.
- New: exported function **IupExecute**. Called by **IupHelp**.
- New: SCROLLTO and SCROLLTOCHILD attributes for **IupScrollBox**.
- New: MARKWHENTOGGLE attribute for **IupTree**.
- New: DELCONTROL attribute for **IupNormalizer**.
- New: **IupStringCompare** utility function to compare strings lexicographically. Used internally in **IupMatrixEx**.
- New: MONITORSCOUNT global attribute.
- New: GLOBALLAYOUTRESIZEKEY global attribute to enable the global keys Ctrl+'+' and Ctrl+'-' that change the FONTSIZE and refresh the layout of the dialog. If element sizes are NOT set using RASTERSIZE their sizes will be automatically increased and decreased.
- New: **IupAnimatedLabel** control.
- New: "IUP_CircleProgressAnimation" pre-defined animation in **IupImageLib** to be used in **IupAnimatedLabel** to show indefinite progress.
- New: functions **IupLoadAnimation** and **IupLoadAnimationFrames** in IUP-IM utilities library.
- New: PASTEFILEAT attributes for **IupMatrixEx**.
- New: MASKNOEMPTY attribute of **IupList** and **IupText**.
- New: **IupCalendar** and **IupDatePick** controls.
- New: date parameter in **IupGetParam**.
- New: CARETCOLOR, CARETSTYLE and CARETWIDTH attributes in **IupScintilla**.
- **Changed:** **IupGetText** pre-defined dialog function API to include the maximum string size. This was a security fix.
- **Changed:** the global keys Alt+Ctrl+Shft+L to display the **IupLayoutDialog** now needs the global attribute GLOBALLAYOUTDLGKEY to be enabled. This was a security fix.
- **Changed:** Scintilla updated to version 3.6.2. Removed KEYSUNICODE attribute from **IupScintilla**, not supported in Scintilla anymore.
- **Changed:** line breaks in a param tip on **IupGetParam** dialog using the '\r' character.
- **Changed:** removed SCROLLTO and SCROLLTOPOS attributes from **IupScintilla**. They were incorrect and not available in Scintilla. New attribute SCROLLBY.
- **Changed:** MASK*L:** and MASK**:C* options for MASK attribute in **IupMatrix**.
- Fixed: defaultenter and defaultesc when dialog has an **IupMatrix** element.
- Fixed: sorted line indices when line is added or removed in **IupMatrixEx**. Now when lines are added or removed the sorting is disabled and must be set again manually.
- Fixed: toggle cells being changed when READONLY=Yes in **IupMatrix**.
- Fixed: progressbar update in **IupProgressDlg** for fast processing.
- Fixed: iup.**dofile** and iup.**dostring** in Lua to process multiple return values and leave them on the stack.
- Fixed: **IupSetHandle** and **IupGetName** usage of the internal cache for names. And name search optimized. Now when a name is set, the control will have a HANDLENAME attribute with the last name set.
- Fixed: LASTFILENAME in **IupMatrixEx** for the import dialog. It is now also used as the initial FILE attribute.
- Fixed: image transparency in **IupItem** when using Visual Styles in Windows.
- Fixed: case insensitive search in **IupMatrixEx**.
- Fixed: old name VISIBLE_ITEMS in **IupList** on Windows.
- Fixed: SELECTION attribute in **IupScintilla**.
- Fixed: dropdown list VALUE can now be set to NULL inside DROP_CB in **IupMatrix**.
- Fixed: AUTOHIDE behavior in **IupSplit**.
- Fixed: dynamic tab close in **IupTabs** on GTK.
- Fixed: missing iup.**GLSizeBox** in Lua.
- Fixed: Lua registration of iup.**submenu** and iup.**user** as containers.
- Fixed: **iuplua_pushihandle** when the control was created in C and it is a container.
- Fixed: dialog must be mapped before IUP_GETPARAM_INIT in **IupGetParam**.
- Fixed: export data when value contains the separator in **IupMatrixEx**. Import data when value contains double quotes (") at start and end, and contents may contains the separator.
- Fixed: IMAGEPOSITION in **IupFlatButton**.
- Fixed: COUNT attribute in **IupText** on GTK.

Version 3.16 (15/Sep/2015)

- New: header "iup_plus.h" with the first version of the C++ API.
- **New:** NOHIDESEL attribute for **IupText** on Windows. The default is Yes so it changed the default behavior of the control that was hiding the selection when loses focus, now by default the selection is always visible just like the **IupList** and the **IupTree**.
- **New:** TXTHLCOLOR global attribute.
- **New:** HLCOLOR attribute for **IupTree** on Windows and Motif (background of selected nodes). The selection color now will not change when the control loses its focus on Windows.
- **New:** LINEALIGMENT*L* and ALIGN*L:C* attributes for **IupMatrix**.
- **New:** MARKATTITLE attribute for **IupMatrix**.
- **New:** IMAGEAUTOSCALE and IMAGESDPI global attributes, AUTOSCALE attribute for **IupImage.**
- **New:** global attributes GL_VERSION, GL_VENDOR and GL_RENDERER, available only after the first call to **IupGLMakeCurrent**.
- **New:** iup.**GetParamHandle** utility function in Lua for **IupGetParam** parameters.
- **New:** SHOWDIALOG and SORTLINEINDEX*id* attributes in **IupMatrixEx**.
- **New:** RETRYCANCEL and YESNOCANCEL button configurations for **IupMessageDlg**.
- **New:** Improved **IupLuaConsole** application with a new command line and console output without using standard output and standard input.
- **New:** DEFAULTBUTTONPADDING global attribute to control the default padding in pre-defined dialogs.

- **New:** TOGGLEIMAGEON/TOGGLEIMAGEOFF, SORTIMAGEDOWN/SORTIMAGEUP and DROPIMAGE attributes for **IupMatrix**. Default images improved with a clear design.
- **New:** TOGGLECENTERED attribute for **IupMatrix**, to center the toggle and use the cell value in place of TOGGLEVALUE*L:C*. No text will be drawn.
- **New:** STATEREFRESH attribute for **IupExpander**.
- **Changed:** HLCOLOR attribute that defines an overlay color for the selected cells in **IupMatrix**. The default is the TXTHLCOLOR global attribute.
- **Changed:** mouse wheel processing will now occur also when the canvas is not in focus while the cursor is over the canvas in Windows.
- **Changed:** although callbacks implemented as simple strings to be executed are still valid in IupLua, returns in strings are not accepted anymore.
- **Changed: IupFlatButton** with TOGGLE=Yes, **IupGLToggle**, and **IupToggle** when a child of an **IupRadio**, and **IupZbox** immediate children will now automatically receive a handle name.
- **Changed:** IupImageLib now contains less images in its pre-compiled library, because we increased the image size to 32x32 with 32bpp in Windows. Images will be automatically resized if necessary using the IMAGESTOCKSIZE global attribute, its default value depends on the screen resolution.
- **Changed:** OPACITY and OPACITYIMAGE **IupDialog** attributes behavior in Windows so they can be set before map as GTK.
- **Changed:** Lua pre-compiled binaries are now separated by folders Lua51/Lua52/Lua53.
- **Changed:** distribution packages are now split according to the Lua version.
- **Changed:** renamed AXS_AUTOSCALEEQUAL to AXS_SCALEEQUAL in **IupPlot**. Old name still works. Now it does not depends on automatic scaling anymore.
- **Changed:** attribute SHOW_TEXT in **IupGauge** and **IupGLProgressBar** renamed to SHOWTEXT. EDIT_MODE, FOCUS_CELL renamed to EDITMODE, FOCUSCELL in **IupMatrix**. VISIBLE_ITEMS renamed to VISIBLEITEMS in **IupList**. Old names still supported.
- **Changed: IupPopup** now can turn an already visible dialog into a modal dialog and interrupt processing. A call to **IupShowXY** for a modal dialog will now update its position.
- **Fixed: IupConfigDialogShow** behavior when dialog is not resizable. It will also do no adjustments if the dialog is already visible. Better control of the maximize of the first time.
- **Fixed:** error report in Lua to always include the traceback.
- **Fixed:** inactive button being activated by mnemonic.
- **Fixed:** CURSORPOS global attribute set when Taskbar is at left or top in Windows.
- **Fixed:** RESTORE attribute in **IupDeatchBox** when old brother is not a child of old parent anymore. Added the possibility of using a different parent with value. Improved the initial new dialog size to use the current size of the child.
- **Fixed:** BUTTON_CB callback in **IupButton** on Windows when the dialog is destroyed.
- Fixed: removed luaL_register dependency from Lua >= 5.2 bindings.
- Fixed: MULTILINE attribute in **IupMatrix** to include vertical scrollbars.
- Fixed: **IupToggle** natural size when has a check box on Windows for DPI aware applications.
- Fixed: support for Windows 10 in global attributes and in the Manifest file.
- Fixed: support for DPI aware in the Manifest file on Windows. This also solves the problem of blur interfaces when using the system DPI Scaling.
- Fixed: VALUE attribute for **IupRadio** in Lua.
- **Fixed:** OPACITYIMAGE attribute in **IupDialog** on GTK 2.x.
- **Fixed:** behavior of AXS_AUTOSCALEEQUAL/AXS_SCALEEQUAL in **IupPlot** when in Zoom.
- **Fixed:** sort item in context menu not visible when read-only in **IupMatrixEx**. Sort sign not being erased correctly when SORTCOLUMNid was set.
- **Fixed:** missing PARAM_CB callback in Lua for **IupParamBox**.
- **Fixed: IupColorDlg** and **IupFontDlg** dialogs in 32 bits on Windows.
- **Fixed:** SORTCOLUMNCOMPARE_CB callback in **IupMatrixEx**.
- **Fixed:** global attribute LANGUAGE when set after the control classes being registered in **Iup*Open** functions.
- **Fixed:** coordinates in WHEEL_CB callback in **IupCanvas** when using multiple monitors on Windows.
- **Fixed:** interference of resize column and selection in **IupMatrix**.
- **Fixed:** finding the letter "M" in **IupMatrixEx** find dialog.
- **Fixed:** iup.isprint in Lua.
- **Fixed:** ICON attribute to be able to use an image from IupImageLib in **IupDialog.**
- **Fixed:** BARSIZE when set to 0 in **IupSplit**. Improved support for AUTOHIDE.
- **Fixed:** ACTIVE attribute when the element is inside a dialog that is disabled by another dialog popup.

Version 3.15 (06/Jul/2015)

- New: "iup_class_cbs.hpp" header with macros to help creation of callbacks as methods in C++.
- New: SHOWNOACTIVE and SHOWMINIMIZENEXT attributes to control the PLACEMENT behavior of the **IupDialog** in Windows.
- New: global hot key (Alt+Ctrl+Shft+L) to show the current dialog layout in a **IupLayoutDialog** dialog.
- New: ELAPSEDTIME attribute available inside the ACTION_CB callback of the **IupTimer**.
- New: Tutorial section in the documentation. It is still under construction but already has several topics completed.
- New: **IupFlatButton** control that mimics a **IupButton** but does not have native system decorations.
- New: FGCOLOR display in rectangle when TITLE and IMAGE are both not defined in **IupGLButton**.
- New: **IupConfig** support in Lua.
- **Changed:** the DESTROY_CB callback to be called before the LDESTROY_CB callback, so it can be processed also in Language bindings.
- **Changed:** the color selection in **IupFontDlg** on Windows is now hidden by default. To show it must define SHOWCOLOR=Yes attribute. The Script selection since was not being used is now always hidden.
- **Changed:** improved documentation in PDF, now with table of contents.
- **Changed:** internal organization in IupLua for better nomenclature.
- **Changed:** renamed iup.TreeSetDescentsAttributes to iup.TreeSetDescendantsAttributes.
- **Fixed:** invalid lin,col at EDITION_CB when mode=1 in **IupMatrix**.
- **Fixed:** layout update to **IupMatrix** internal children.
- **Fixed:** drag&drop of tree nodes in **IupTree** on Windows. (Thanks to Nodir T.)
- **Fixed: IupPlot** destroy (affected MingW). (Thanks to Ducan G.)
- **Fixed: IupPopup** when used for the last visible dialog.
- **Fixed:** size computation in **IupToggle** to not include the spacing between the check box and the text, if text is NULL or empty.
- **Fixed:** TABIMAGE attribute when resetting to NULL in **IupTabs**.
- **Fixed:** scrollbars when *AUTOHIDE=Yes in **IupCanvas** on Windows.
- **Fixed: IupExpander** animation timing.
- **Fixed:** FITTOSIZE attribute in **IupMatrix** to consider the BORDER attribute.

Version 3.14 (28/Apr/2015)

- New: support for Lua 5.3.
- New: INFOTIP attribute for **IupTree** on Windows.
- New: OPENCOLOR and HIGHCOLOR attributes for **IupExpander** to change the title text color in different situations. New TITLEIMAGE* attributes to use a title image instead of a text. New **TITLEEXPAND** attribute to enable expand/contract action in the title.
- New: ANIMATION attribute for **IupExpander** to enable animation during open/close.
- New: MULTIVALUECOUNT and MULTIVALUE*id* attributes for **IupFileDlg** when MULTIPLEFILES=Yes.
- New: AXS_AUTOSCALEEQUAL and VIEWPORTSQUARE attributes for **IupPlot**.
- New: CHILDOFFSET attribute for native containers (**IupTabs**, **IupFrame**, **IupDialog**, **IupBackgroundBox**, **IupScrollBox**).
- New: CELL attribute in **IupMatrix** to return the displayed value.
- New: EDITFITVALUE, EDITVALUE, EDITTEXT, EDITALIGN, EDITHIDEONFOCUS attributes for **IupMatrix** to control editing cell values and focus.
- New: EDITCLICK_CB, EDITRELEASE_CB and EDITMOUSEMOVE_CB callbacks for **IupMatrix** called when EDITHIDEONFOCUS=NO and editing is on going right before CLICK_CB, RELEASE_CB and MOUSEMOVE_CB callbacks.
- New: CELLNAMES attributes for **IupMatrix** when using formulas.
- **Changed: IupMap** will now call **IupRefresh** when mapping the dialog after all other processing. This affects the MAP_CB callback of the children, that it will be called before the layout is updated, so the children current size will still be 0x0 during MAP_CB.
- **Changed: IupExpander** internally remodeled to use other IUP elements to compose its handler area. The controls can be accessed by IupGetChild* and then reconfigured if necessary. There are no size limitations for images anymore. Extra buttons are now creation-only.
- **Changed:** when editing a cell value the caret will now be positioned closest to where the user double clicked in **IupMatrix**.
- Fixed: visual feedback when interactively moving the title or the legend box in **IupPlot**.
- Fixed: transparency for images in **IupTabs** on Windows. (Thanks to Nodir T.)
- **Fixed:** COPYDATA_CB callback for Unicode support in **IupDialog** on Windows. (Thanks to Nodir T.)
- **Fixed:** setting TITLE attribute to NULL in **IupPlot.**
- **Fixed:** automatic margin calculation in **IupPlot** when TITLE is not defined, top margin is automatic and bottom margin is manually set.
- **Fixed:** Fixed FGCOLOR and FONT, when set before map in **IupScintilla**.
- **Fixed:** update of CMARGIN and CGAP attributes when FONT is changed in **IupGridBox**, **IupHbox**, and **IupVbox**.
- **Fixed:** resize of a **IupButton** with just a color on GTK.
- **Fixed:** ACTION callback for **IupBackgroundBox** in Lua.
- **Fixed:** ENTERWINDOW_CB/LEAVEWINDOW_CB call order in Windows.
- **Fixed:** memory leaks in drag&drop processing in Windows. Memory leaks in internal **IupImage** cache.
- **Fixed:** Enter and Esc keys behavior in **IupList** when DROPDOWN=Yes on Windows, while the dropdown list is shown they must simply close the list and do not forward the action to the

dialog.
- **Fixed:** VALUE attribute during VALUECHANGED_CB in **IupList** when EDITBOX=Yes on Windows and an item is selected with the keyboard.
- **Fixed:** add new units of an existing quantity in **IupMatrixEx**.
- **Fixed:** font parsing when using old invalid names.
- **Fixed: range** and **cell** functions inside formulas when reference another cell that also uses formulas in **IupMatrix**. Recurrence in **range** and **cell** functions.
- **Fixed:** DIRECTORY attribute return value when MULTIPLEFILES=Yes in **IupFileDlg** on Windows and GTK.
- **Fixed:** toggle visible state when moving or copying a node in **IupTree** in Windows.
- **Fixed:** keys processing in **IupMatrixEx** that were disabling **IupMatrix** regular processing.
- **Fixed:** column resize feedback for **IupMatrix** when in GTK3.
- **Fixed:** CLEAR attribute in **IupPlot**.

Version 3.13 (04/Feb/2015)

- New: global attribute DEFAULTFONTFACE.
- New: EXPANDVERTICAL and EXPANDHORIZONTAL for **IupGLCanvasBox** children.
- New: XHIDDEN and YHIDDEN attributes for scrollbar information in **IupCanvas**.
- New: TIPFORMAT, AXS_*TIPFORMAT, TITLEPOS and DS_USERDATA attributes in **IupPlot**. New value "XY" for LEGENDPOS attribute. New attribute MENUITEMPROPERTIES and new Properties Dialog. New attributes GRIDMINOR, GRIDMINORCOLOR, GRIDLINEWIDTH and GRIDLINESTYLE. New attributes AXS_*TICKROTATENUMBERANGLE, AXS_*TICKFORMATAUTO and AXS_*TICKFORMATPRECISION. New attributes BACKIMAGE, BACKIMAGE_XMIN, BACKIMAGE_XMAX, BACKIMAGE_YMIN, BACKIMAGE_YMAX.
- New: **IupPlotFindSample**, **IupPlotAddSegment, IupPlotInsertSegment** auxiliary functions for **IupPlot**.
- New: **IupPlotSetFormula** auxiliary function for **IupPlot**.
- New: NACTIVE attribute for the **IupDialog**.
- New: CELL_EDITED attribute set during VALUE_EDIT_CB and VALUECHANGED_CB when the cell was interactively changed in **IupMatrix**.
- New: **IupMatrixSetFormula** and **IupMatrixSetDynamic** auxiliary functions for **IupMatrix**.
- New: TRANSLATEVALUE_CB callback for **IupMatrix**.
- New: EDITING state attribute for **IupMatrix**.
- New: **IupParamf** and **IupParamBox** utility functions, exported from **IupGetParam** internals.
- New: CELLBYTITLE attribute for **IupMatrixEx**. Affects Go To and Copy To dialogs.
- **Changed:** IupLua console file selection to include filter "*.lua".
- **Changed: IupLayoutDialog** context menu to include "Set Focus" and "Blink" items.
- **Changed:** processing of MENUCONTEXT_CB callback return value to accept IUP_IGNORE in **IupMatrixEx**.
- **Changed: IupPlotInsertPoints, IupPlotInsertStrPoints, IupPlotAddPoints, IupPlotAddStrPoints**, renamed to **IupPlotInsertSamples, IupPlotInsertStrSamples, IupPlotAddSamples, IupPlotAddStrSamples.**
- **Changed:** SUNKEN attribute in **IupFrame** is not creation only anymore.
- **Changed:** MathGL updated to version 2.3.2.
- **Changed:** Scintilla updated to version 3.5.3.
- Fixed: **IupScanf** when maximum number of characters allowed is reached by the data in a given variable.
- Fixed: **IupGLCanvasBox** mouse coordinates processing.
- Fixed: **IupGLSubCanvas** font processing.
- Fixed: scrollbar programmatic update crash in **IupCanvas** on Windows.
- Fixed: remove of a hidden tab in **IupTabs** on Windows.
- Fixed: separator in **IupGetParam**.
- Fixed: WHEEL_CB in **IupCanvas** on GTK3.
- Fixed: paste of empty cell in **IupMatrixEx**.
- Fixed: AXS_XTICKFORMAT and AXS_YTICKFORMAT attributes when defined by the application in **IupPlot**.
- Fixed: global attribute CURSORPOS when Start Menu is positioned at left or top of the screen in Windows.
- Fixed: **IupShow** when redisplaying a dialog without changing its position when Start Menu is positioned at left or top of the screen in Windows.
- Fixed: sorting of numeric values in **IupMatrixEx**.
- Fixed: VALUE attribute inside PARAM_CB callback when user click in auxiliary buttons for Font, Color or File Selection in **IupGetParam**.

Version 3.12 (19/Nov/2014)

- New: **IupConfig*** functions to manage application configuration files.
- New: **IupPlot** control, that will replace **IupPPlot**. It eliminates all the limitations and issues, improves interaction a lot and uses double instead of float.
- New: VALUESTRING attribute for **IupList**.
- New: USERSIZE attribute for all elements.
- New: SHOWCONTEXTMENU*L:C* attribute for **IupMatrixEx**.
- New: NUMERICDECIMALSYMBOL attribute for **IupMatrixEx**.
- New: Dialog to configure TEXTSEPARATOR, NUMERICFORMATPRECISION and NUMERICDECIMALSYMBOL in **IupMatrixEx** context menu.
- New: FILEDIRECTORY and LASTFILENAME attributes for Export and Import file dialogs in **IupMatrixEx** context menu.
- New: MENUCONTEXTCLOSE_CB callback in **IupMatrixEx**.
- New: SKIPLINES and SKIPCOLUMNS option attributes used in COPYFILE export action in **IupMatrixEx**.
- **New:** OPACITYIMAGE attribute for **IupDialog** in Windows and GTK.
- **New:** VALUECHANGED_CB callback for **IupSplit**.
- **Changed:** TEXTSEPARATOR attribute is now used also for all COPY* attributes in **IupMatrixEx**. Removed TEXTNUMERICLOCALE attribute. TEXTFORMAT renamed to FILEFORMAT.
- **Changed:** FONTFACE is not read-only anymore.
- **Changed:** default font typeface changed to Helvetica in **IupGLSubCanvas**.
- **Changed: IMPORTANT** - GTK defaults to GTK version 3 starting at Linux 3.13. (Notice no XOR support in CD)
- Fixed: number separator in COPY* and PASTE* attributes in **IupMatrixEx**.
- Fixed: **IupTabs** VALUE/VALUEPOS attributes after tabs were dynamically inserted in Windows.
- Fixed: VALUECHANGED_CB in **IupScintilla**. ZOOM_CB declaration in Lua.
- Fixed: automatic inactive appearance for images with 24bpp in Windows.

Version 3.11.2 (06/Oct/2014)

- New: TITLEBACKIMAGEINACTIVE attribute for **IupGLFrame** and **IupGLExpander**.
- New: BACKIMAGEINACTIVE attribute for **IupGLFrame**.
- New: BACKIMAGE* attributes for **IupGLButton, IupGLVal** and **IupGLProgress**.
- New: FRONTIMAGE* attributes for **IupGLButton**.
- **New:** FITTOBACKIMAGE attribute for **IupGLButton, IupGLVal** and **IupGLProgress**.
- New: global attribute DEFAULTPRECISION that affects how real values are shown by default in **IupGetParam** and **IupMatrixEx**.
- New: attribute EMPTYAS3STATE for **IupTree** on Windows.
- New: value AREA for DS_MODE attribute in **IupPPlot**.
- **Changed:** all images in **IupGLControls** are now drawn using OpenGL textures instead of glDrawPixels.
- Fixed: image complementary attributes like IMAGEPRESS, IMAGEHIGHLIGHT, IMAGEINACTIVE, and others, in **IupGLControls**.
- Fixed: BGCOLOR_CB and FGCOLOR_CB callbacks of **IupMatrix** in Lua when an invalid number of arguments is returned. (Thanks to Alex M.)
- Fixed: parsing of non-numeric values in **IupMatrixEx** when column has numeric values.
- Fixed: pressed feedback in **IupGLToggle** when using IMAGEPRESS.
- Fixed: image draw using OpenGL textures in **IupGLControls** was upside down, and now does not depends on the background color anymore.
- Fixed: **IupGLExpander** behavior when BARPOSITION is BOTTOM or RIGHT.
- Fixed: **IupGLVal** interaction when IMAGE is defined.
- Fixed: elements positioning when BORDER=Yes in **IupGLControls**.
- Fixed: characters processing for the ACTION callback in **IupText** when they are generated using an AltGr key combination (Ctrl+Alt) on Windows.
- Fixed: Ctrl+A key combination in **IupText** when AltGr key is used on Windows.
- Fixed: all **IupSet*Id** functions when the element actually does not support **Id** based attributes but the application use the function for custom attributes.
- Fixed: scrollbar programmatic update in **IupCanvas** on Windows.

Version 3.11.1 (01/Sep/2014)

- New: COMCTL32VER6 global attribute that informs if the Windows common controls are using Visual Styles or not.
- New: SHOWGRIP option to change grip in **IupSplit** for a double continuous line using LINES value. Also when SHOWGRIP=NO and COLOR is defined the grip area is filled with the color.
- New: **IupMglLabel** based on **IupMglPlot** so TeX symbold can be displayed without the need for a plot.
- New: TITLEBACKIMAGE and BACKIMAGE attributes for **IupGLFrame**. TITLEBACKIMAGE attribute for **IupGLExpander**.

- New: MOVETOTOP attribute for **IupGLFrame** and **IupGLExpander**.
- New: **IupGetDouble*** and **IupSetDouble*** functions.
- New: double parameters in **IupGetParam**. PRECISION attribute to control real values when interactively changed.
- **Changed: IMPORTANT** - repository migrated from CVS to SVN.
- Fixed: missing iup.**ImageFromImImage** in Lua.
- Fixed: **IupImageFromImImage** when flipping bottom-top to top-bottom. (Thanks to Jeremiah N.)
- Fixed: support for global menu in recent Ubuntu systems.
- Fixed: Font support in **IupGLControls** when UTF8MODE=Yes.
- Fixed: DEFAULTFONT processing. It was affecting FONT inheritance.
- Fixed: CARET attribute in **IupScintilla**.
- Fixed: **IupGLFrame** natural size computation.
- Fixed: MOVEABLE attribute in **IupGLCanvasBox** when moving native based elements like IupText and IupMatrix.
- Fixed: **iup.glprogressbar** creation in Lua.
- Fixed: improved support for liboverlay-scrollbar in Ubuntu, but SCROLL_CB is limited to IUP_SBPOSV and IUP_SBPOSH codes.
- Fixed: **IupDialog** layout update when maximizing the window on GTK. Invalid IupFlush removed from **IupCanvas** layout update on GTK.
- Fixed: DRAWABLE attribute inside FILE_CB callback in **IupFileDlg** on GTK.
- Fixed: Added compatibility code for GTK 3.10.
- Fixed: **IupFrame** background color on GTK version 3.x
- Fixed: default values that were dependent on the current locale. DENSITY in **IupDial**, DX and DY in **IupCanvas**, STEP and PAGESTEP in **IupVal**, and several in **IupMglPlot**.
- Fixed: Management of hidden tabs in **IupTabs** on Windows.
- Fixed: LINEVALUE attribute return value in **IupScintilla**. CARET attribute in **IupScintilla** when "lin" is greater than the last line.
- Fixed: ZORDER attribute for **IupGLControls** elements.
- Fixed: **IupButton** and **IupToggle** mouse over feedback when not using Visual Styles and FLAT=Yes on Windows.

Version 3.11 (28/Jul/2014)

- New: CD_IUPDBUFFER and CD_IUPDBUFFERRGB drivers in the **iupcd** library. **IMPORTANT:** This IUP version depends on CD version 5.8.
- New: IMAGE, IMHIGHLIGHT, IMOPEN, and IMOPENHIGHLIGHT attributes for replacing the arrow, or the arrow and the title of a **IupExpander** when BARPOSITION=TOP.
- New: EXTRABUTTONS, IMAGEEXTRAid, IMAGEEXTRAPRESSid, IMAGEEXTRAHIGHLIGHTid attributes and EXTRABUTTON_CB callback for **IupExpander** to handle extra buttons at right when BARPOSITION=TOP.
- New: **IupPPlotGetSample** and **IupPPlotGetSampleStr** functions for **IupPPlot**.
- New: LOADLEXERLIBRARY attribute in **IupScintilla**.
- New: DRAGCURSOR attribute for the Drag & Drop support.
- New: SWAPBUFFERS_CB callback for **IupGLCanvas**.
- New: COLORUPDATE_CB callback for **IupColorDlg**.
- New: HORIZONTALFREE and VERTICALFREE values for the EXPAND attribute. The element will simply expand to the available free space at the container, and it will not affect the container expand.
- New: DEFAULTFONTSTYLE global attribute.
- New: OPENCLOSE_CB callback for **IupExpander**.
- New: IupGLControls an OpenGL embeddable controls library.
- New: **IupScintillaSendMessage** function.
- **Changed:** SEPARATOR attribute behavior in **IupLabel** will now use the HORIZONTALFREE and VERTICALFREE values for the EXPAND attribute.
- **Changed:** updated Scintilla version to 3.4.4.
- **Changed:** all the controls in the additional controls library (matrix, colorbrowser, gauge, dial, etc) and the IupPPlot control now uses the new CD_IUPDBUFFER* drivers.
- **Changed: IupMglPlot** API replaced "float" by "double". Removed support for TrueType (*.ttf) and OpenType (*.otf) font files. PLANARVALUE and CLOUDCUBES attributea are not supported anymore. AXS_?MIN attributes default changed to -1, to match MathGL default. MathGL updated to version 2.2. **IupMglPlotTransformXYZ** renamed to **IupMglPlotTransformTo**. Plor area configured using boolean atributes MARGINLEFT, MARGINRIGHT, MARGINTOP, and MARGINBOTTOM.
- Fixed: SORTSIGN attribute in **IupMatrix** when set to NO.
- Fixed: first item in single selection **IupTree** was not showing that it is selected.
- Fixed: **IupScrollBox** when there is no child.
- Fixed: LISTACTION_CB callback when state=0 in **IupMatrixList**.
- Fixed: ENTERWINDOW_CB callback for **IupLabel** on Windows.
- Fixed: BGCOLOR support in **IupBackgroundBox**.
- Fixed: **IupExpander** expansion when closed. Alignment of arrow and title when bar size is greater than default.
- Fixed: support for icons with multiple sizes in **IupDialog** on Windows.
- Fixed: delete all items in **IupMatrixList** when click the del button on title line.
- Fixed: VALUE attribute of **IupList** when EDITBOX=YES and the text box is empty on Windows.
- Fixed: MASK attribute processing in **IupMatrixList** when starting to edit a label.
- Fixed: Ctrl+V key combination to Paste cells starting at the focus cell in **IupMatrixList**.
- Fixed: **IupVal** behavior when inside an **IupBackgroundBox** or **IupScrollBox** on Windows.
- Fixed: attribute return value in Lua when it was not a string.
- Fixed: **IupFill** so it can be placed inside a **IupGridBox**, the behavior will be the same as inside an **IupHbox**.
- Fixed: FITTOCHILDREN attribute in **IupGridBox**.
- Fixed: **IupPPlotTransformTo** missing from DLLs and from Lua.
- Fixed: Ctrl and Shift keys scrolling the **IupMatrix** when pressed.
- Fixed: SPIN_CB callback in **IupText** when value was incremented/decremented out of range.
- Fixed: CLIENTSIZE attribute in **IupDetachBox**, **IupSbox**, **IupSpin** and **IupSplit**. Changed CLIENTSIZE in **IupHbox**, **IupVbox** and **IupGridBox** to use only the Current size.
- Fixed: CLIENTOFFSET attribute in **IupBackgroundBox**.
- Fixed: K_ANY being called twice for each key in **IupScintilla**.
- Fixed: COUNT attribute return value in **IupText** on Windows.
- Fixed: dynamic insert of the first Tab in **IupTabs** on Windows. Dynamic remove of the last Tab in **IupTabs**. Management of hidden tabs in **IupTabs** on Windows.
- Fixed: NAME cache attribute when element is removed.

Version 3.10.1 (24/Jan/2014)

- New: RESIZEMATRIX_CB callback in **IupMatrix**.
- New: internal **IupMatrix** callbacks BUTTON_CB, MOTION_CB and KEYPRESS_CB are now exported to Lua as "MatButtonCb", "MatMotionCb" and "MatKeyPressCb".
- New: AUTOCSELECTION_CB, AUTOCCANCELLED_CB and AUTOCCHARDELETED_CB callbacks in **IupScintilla**.
- Fixed: key names in IupLua when modifiers are used. (Thanks to kmx)
- Fixed: text with computation in UTF-8 on Windows. (Thanks to kmx)
- Fixed: 3 state check box size in **IupTree** on Windows when using Classic Style. (Thanks to kmx)
- Fixed: IUP_MOUSEPOS on Windows when the taskbar is at the top, or left of the screen.
- Fixed: TABTITLE return value after tabs are added or removed.
- Fixed: **IupBackgroundBox** creation in Lua.
- Fixed: map error in **IupTabs** on GTK.

Version 3.10 (17/Jan/2014)

- New: DRAGDROPTREE attribute to enable automatic drag&drop between **IupTree**s in the same application.
- New: DRAGDROPLIST attribute to enable automatic drag&drop between **IupList**s in the same application.
- New: SHOWCLOSE attribute and TABCLOSE_CB callback for **IupTabs** to show a close button in each tab.
- New: RIGHTCLICK_CB callback for **IupTabs**.
- New: TASKBARPROGRESS, TASKBARPROGRESSSTATE and TASKBARPROGRESSVALUE attributes for **IupDialog** on Windows to show a progress feedback on the taskbar running on Windows 7+.
- New: COPY, SELECTALL, PRINT and ZOOM attributes in **IupWebBrowser**.
- New: function **IupImageFromImImage** in the IUP-IM library.
- New: FITTOCHILDREN attribute for **IupGridBox**.
- New: **IupDetachBox** container element to allow interactively detach of an element and insert it in a new dialog.
- New: TEXTNUMERICLOCALE attribute for **IupMatrixEx** to allow a different locale during paste of numeric values.
- New: **IupBackgroundBox** native container to allow more control of children visibility.
- **Changed:** "IUP_EditErase" image to use "gtk-delete" definition instead of "gtk-close" in IupImageLib.
- **Changed:** TABIMAGEn and TABTITLEn attributes in **IupTabs** to update the respective child attribute.
- Fixed: VALUE_HANDLE attribute in **IupZbox** was write-only.
- Fixed: missing **IupMatrixList** Lua binding.

- Fixed: SAVEUNDER attribute in **IupDialog** on Windows.
- Fixed: **IupWebBrowser** on GTK.
- Fixed: horizontal frame color in **IupMatrix**.
- Fixed: **IupLoadBuffer**.
- Fixed: TIP attribute on Windows when not using Visual Styles.
- Fixed: EXPANDCHILDREN in **IupHbox**, **IupVbox** and **IupGridBox** when children contains an **IupFill**.
- Fixed: Caps Lock processing on Windows.
- Fixed: COUNT, APPENDITEM, INSERTITEMid and REMOVEITEM in **IupMatrixList** when EDITABLE=Yes. Drawing of the empty line when there is no other items. Insertion of the empty line when there is less than 2 items in the list.
- Fixed: ACTION callback in **IupText** when using UTF-8.
- Fixed: SCROLLTO and SCROLLTOPOS attributes in **IupText** on Windows when FORMATTING=Yes.
- Fixed: support for UTF-8 in **IupClipboard** on Windows that affected paste in **IupText**.

Version 3.9 (22/Nov/2013)

- New: KEYSUNICODE attribute for **IupScintilla** on Windows.
- New: support for command line processing in the **IupView** application to convert image files to source code that creates an **IupImage**.
- New: IupProgressDlg pre-defined dialog.
- New: utility functions **IupSetInt***, **IupSetFloat***, **IupSetRGB*** and **IupGetRGB***.
- New: support for UTF-8 strings in the Windows and GTK driver using the UTF8MODE global attribute.
- New: ACTION callback for **IupExpander**.
- New: COLRESIZE_CB callback in **IupMatrix**.
- New: COPYCOL, COPYLIN, MOVECOL and MOVELIN attributes in **IupMatrix**.
- New: FRAMETITLEHIGHLIGHT and ALIGNMENTLIN0 attributes in **IupMatrix**.
- New: TYPE*L:C* attribute and TYPE_CB callback in **IupMatrix** that allow to display a color, a progress bar, and an image in a cell.
- New: RESIZEMATRIXCOLOR attribute in **IupMatrix** to control the resize column feedback color.
- New: TOGGLEVALUE attribute and TOGGLEVALUE_CB callback in **IupMatrix** to enable a toggle button inside a cell.
- New: control IupMatrixList that shows a list using an **IupMatrix**.
- New: IupMatrixEx library with an extension package for **IupMatrix**.
- New: VALUECHANGED_CB callback for **IupMatrix**.
- New: **IupSetLanguagePack**, **IupGetLanguageString** and **IupSetLanguageString** functions to help in application Internationalization. Strings starting in "_@" will be automatically retrieved from the internal string database.
- New: MASKFAIL_CB callback for **IupText** and **IupList** when MASK is used and an invalid text is typed.
- New: Ihandle* parameter for **IupGetParam**.
- New: attributes REMOVE and CURRENT in **IupPPlot** now also accepts the DS_NAME as value when setting.
- New: PLOT_COUNT, PLOT_NUMCOL, PLOT_CURRENT, PLOT_INSERT and PLOT_REMOVE attributes for **IupPPlot** to support multiple plots in the same display area.
- New: PLOTBUTTON_CB and PLOTMOTION_CB calbacks for **IupPPlot.**
- **Changed:** preserve of FRAMEVERTCOLOR*, FRAMEHORIZCOLOR*, SORTSIGN*, MASK*, WIDTH*, RASTERWIDTH*, HEIGHT* and RASTERHEIGHT* attributes when lines or columns are added or removed in **IupMatrix**.
- **Changed:** all color values in attributes now accepts also the notation "#RRGGBB" in hexadecimal.
- **Changed:** removed Windows 2000 compatibility.
- **Changed:** UTF8AUTOCONVERT global attribute renamed to UTF8MODE with inverted meaning. Old name still supported for compatibility.
- **Changed:** renamed **IupStoreAttribute** to **IupSetStrAttribute**, and **IupSetStrfAttribute** to **IupSetStrf**, old names kept for compatibility.
- **Changed: IupMglPlot** attribute ZOOM to not use "," where floating point values are specified, changed to ":".
- **Changed:** added support for more keys in iupkey.h. New key definitions: K_LSHIFT, K_RSHIFT, K_LCTRL, K_RCTRL, K_LALT, K_RALT, K_NUM, K_SCROLL, K_CAPS and K_diaeresis. Now all GDK and X11 keys are supported, but non defined keys are supported using its hexadecimal value. Modifiers are now a bit value separated from the base key code, which can be obtained using the macro **iup_XkeyBase**. See the "iupkey.h" file for more definitions. **IMPORTANT**: any C/C++ source code that uses the "iupkey.h" definitions MUST be recompiled.
- **Changed:** added support for keyboard selection in **IupMatrix**.
- **Changed:** When LIMITEXPAND=Yes in **IupMatrix** and the scrollbars have *AUTOHIDE=Yes, the maximum size will not include the scrollbars.
- **Changed:** updated Scintilla version to 3.3.5.
- **Changed:** STEREO attribute processing in **IupGLCanvas** to avoid failure during canvas creation. If stereo is not available it will still create a regular OpenGL context.
- **Changed:** ENTERITEM_CB callback in **IupMatrix** is now also called when focus is changed because lines or columns were added or removed.
- Fixed: repaint of the **IupOleControl** that affected the **IupWebBrowser** on Windows.
- Fixed: NODEREMOVED_CB callback for **IupTree**, was providing the wrong userdata in some cases.
- Fixed: **IupScintilla** library build to internally use the Scintilla name space.
- Fixed: K_ANY callback return code processing in **IupScintilla** on Windows.
- Fixed: VALUE and CHARn attributes returned value in **IupScintilla**.
- Fixed: **IupToggle** focus feedback behavior on Windows.
- Fixed: missing **IupGridBox** register in Lua.
- Fixed: fail to update **IupTabs** when a child is removed on Windows.
- Fixed: documentation of internal callback parameters format list.
- Fixed: unmap of **IupTree**, **IupText** and **IupTabs** on Windows.
- Fixed: **IupScrollBox** expand behavior to not depends on children expansion, just like it is not dependent on children size.
- Fixed: focus behavior in **IupScrollBox**, now CANFOCUS=NO.
- Fixed: LEDC processing of **IupSplit** controls.
- Fixed: **IupGetAttributeHandle** not checking at control implementation.
- Fixed: **IupExpander** layout when closed but has a child that can be expanded.
- Fixed: frame color transparency using BGCOLOR for title cells in **IupMatrix**.
- Fixed: **IupTextConvertPosToLinCol** function for **IupMatrix**.
- Fixed: position of dialog using **IupShowXY** when using IUP_LEFT and IUP_TOP, and the taskbar is at left or top on Windows.
- Fixed: fixed internal test for known non string attributes that affected **IupGetAttributes** and iup.**GetAttribute** in Lua.
- Fixed: clipboard and drag&drop data size on GTK.
- Fixed: MASKFLOAT attribute parsing.
- Fixed: CANFOCUS=NO in **IupVal** on Windows.
- Fixed: **IupWebBrowser** when creating and destroying multiple controls on Windows.
- Fixed: **IupPPlotPaintTo** to update plot sizes.
- Fixed: returned value by SCREENPOSITION/X/Y attributes on Windows when the taskbar is at the top, or left of the screen.
- Fixed: **IupScrollBox** mouse respond when there is no scrollbars.
- Fixed: Spin was not being redraw when ACTIVE was changed in **IupText** on Windows.
- Fixed: unmap of **IupScintilla**.

Version 3.8 (08/May/2013)

- **IMPORTANT:** the pre-compiled binaries are compatible only with CD version 5.6.1 pre-compiled binaries.
- New: attribute TOGGLEVISIBLEid for **IupTree** when SHOWTOGGLE=Yes.
- New: attribute TABVISIBLEid for **IupTabs**.
- New: IupLink control that shows a clickable URL.
- New: IupGridBox container to arrange elements in a regular grid.
- New: IupScintilla control that shows a source code text editor based on the Scintilla library.
- New: support for IUP_CONTINUE return code and FILE attribute update inside the FILE_CB callback when status=OK in the **IupFileDlg** dialog.
- New: IupExpander container to interactively control the visibility of a child inside the dialog.
- **Changed:** GTK stock images now uses the same size as the Windows and Motif images in **IupImageLib**
- Fixed: line detection on strings using DOS line breaks (\r+\n).
- Fixed: **IupScrollBox** child expansion when the container is greater than the child natural size.
- Fixed: **IupScrollBox** binding for Lua.
- Fixed: **IupClipboard** on Windows was clearing the clipboard contents every time data was copied.
- Fixed: **IupWebBrowser** for GTK was using an old function call of the internal SDK.
- Fixed: the DIRECTORY attribute was not being updated when a new file filename was selected in **IupFileDlg**.
- Fixed: in a multi-selection **IupTree** the selection callbacks were being called with status=0 when a single item was selected. on GTK the callback were also called when a branch were simply expanded or contracted.
- Fixed: toggle processing on Windows when SHOWTOGGLE=Yes in **IupTree**. Fixed spacing from toggle to image on Windows. Removed support for SHOWTOGGLE=Yes on Motif.
- Fixed: dialog client size computation on Windows when the Win32 API returns an invalid value.
- Fixed: **IupScrollBox** available space computation.
- Fixed: FGCOLOR and PADDING for **IupLabel** when used before map on Windows.

- Fixed: BUTTON_CB, ENTERWINDOW_CB and LEAVEWINDOW_CB callbacks for **IupLabel** on GTK.
- Fixed: underline and strikeout support on GTK.
- Fixed: **IupMatrix** redraw when selecting lines or columns in a matrix with non scrollable lines or columns.
- Fixed: BGCOLOR return value in **IupButton** on Windows.
- Fixed: "Load Image Lib" feature in **IupView** when using GTK.
- Fixed: ZORDER attribute on Motif.

Version 3.7 (29/Nov/2012)

- New: support for GTK 3. The pre-compiled binaries still use GTK 2. See the GTK driver documentation.
- New: layout composition element **IupScrollBox**.
- New: **SHOWDRAGDROP** attribute and **DRAGDROP_CB** callback to support internal drag and drop of items in **IupList**.
- New: support for global callbacks in Lua.
- New: ADDFORMAT, FORMAT, FORMATAVAILABLE, FORMATDATA and FORMATDATASIZE attributes for **IupClipboard**.
- New: TOGGLE option for VALUE attribute in **IupToggle**.
- Fixed: IMAGEid attribute update in **IupList**.
- Fixed: **IupGetParam** callback return value parsing in Lua.
- Fixed: **IupCanvas** size when scrollbars are hidden on Motif.
- Fixed: **IupLabel** missing drag&drop support.
- Fixed: **IupMatrix** on GTK when editing a cell and Esc was pressed.
- Fixed: the return value for POSX and POSY in **IupCanvas** when the respective scrollbar is hidden or disabled.
- Fixed: detection of the minimum size of a child inside **IupSplit**.
- Fixed: **IupCanvas** RESIZE_CB was called recursively when DX or DY attributes were updated during the callback and XAUTOHIDE=Yes or YAUTOHIDE=Yes.
- Fixed: maintain LASTADDNODE id consistent when one or more nodes are removes in **IupTree**.
- Fixed: key processing in **IupText** and **IupList** on Motif to avoid Alt, Ctrl and Sys keys to generate text input.
- Fixed: **IupLabel** mnemonic parsing on GTK.
- Fixed: Mnemonic processing on Windows.
- Fixed: **IupButton** visual feedback when the user double click the button on Windows.
- Fixed: **IupToggle** response when the user double click the button on Windows.
- Fixed: natural size computation in **IupMatrix** when BORDER=Yes.

Version 3.6 (23/June/2012)

- New: Drag&Drop attributes and callbacks for **IupDialog**, **IupCanvas**, **IupText**, **IupList**, and **IupTree**. Old DRAGDROP attribute renamed to DROPFILESTARGET, old still works for compatibility.
- New: CELLBGCOLORL:C and CELLFGCOLORL:C attributes for **IupMatrix**.
- New: MAXSTR attribute for a string parameter in **IupGetParam**. Titles can now contain the '%' character by using two characters "%%". New definitions for the callback parameters when index is negative.
- New: SHOWTOGGLE attribute and TOGGLEVALUE_CB callback for **IupTree**.
- New: DS_COUNT attribute in **IupPPlot**.
- New: **IupMglPlot** element using almost the same interface (attributes and callbacks) as **IupPPlot** but with support for 3D coordinates and many other plot options.
- New: NATURALSIZE attribute for all elements.
- New: support for images in **IupList** items using the SHOWIMAGE and IMAGEid attributes.
- New: ARBCONTEXT, CONTEXTVERSION, CONTEXTFLAGS and CONTEXTPROFILE attributes for **IupGLCanvas**.
- New: **ScriptBasic** Binding by John Spikowski at the SB Forum.
- New: FILTER status in FILE_CB callback for **IupFileDlg** on Windows.
- New: TOUCH attribute for **IupDialog** on Windows.
- New: LASTERROR global attribute on Windows.
- New: parameter in LEDC "-s" to declare image data as static.
- New: TRAYTIPBALLOON, TRAYTIPBALLOONDELAY, TRAYTIPBALLOONTITLE and TRAYTIPBALLOONTITLEICON **IupDialog** attributes on Windows. And TRAYTIPMARKUP on GTK.
- New: global attribute IUPLUA_THREADED so IUP can be used inside coroutines in Lua.
- New: callback MENUDROP_CB for **IupMatrix** to show a popup menu instead of a dropdown list.
- New: support for AZERTY keyboards on Windows.
- New: CLEARVALUE and CLEARATTRIB attributes for **IupMatrix**.
- New: NONE option for the EDITNEXT attribute in IupMatrix.
- Changed: optimized **IupImage** internal cache.
- Changed: removed Lua bytecode usage in pre-compiled binaries. Now IUP pre-compiled binaries are compatible with LuaJIT.
- **Changed:** the MINSIZE and MAXSIZE attributes for **IupDialog** now also behaves as the other elements.
- **Changed:** added internal string limitations for **IupGetParam**, **IupGetFile** and **IupGetText**.
- **Changed: ADDLEAF**id **and ADDBRANCHid attributes in IupTree now accepts -1 to insert a node before the root node.**
- **Changed:** improved performance of APPENDITEM and INSERTITEM in **IupList** on Windows.
- **Changed:** improved mouse edition interaction in **IupMatrix**. Now the edition is started only when left button is released after a double click. Also if DROPCHECK_CB is defined and return IUP_DEFAULT for a cell, to show the dropdown list or the new popup menu the user can simply do a single click in the drop feedback area of that cell.
- **Changed:** REDRAW attribute interval in **IupMatrix** now uses "-" for separator as other attributes. Old separator is still accepted.
- **Changed:** if TEXT or IMAGE attributes set to NULL in **IupClipboard** clears the clipboard data.
- **Changed:** horizontal alignment for text with multiple lines in **IupButton** now will also align each line on Windows.
- **Undo:** removed "P" from **IupPPlot** additional API functions, because they will be used also for other plot controls. Old names still exists for compatibility. The new functions need more flexibility and they must co-exist.
- Fixed: NUMCOL_NOSCROLL and NUMLIN_NOSCROLL attributes for the **IupMatrix** when scrolling with the scrollbar arrows up to the top or to the left.
- Fixed: **IupMatrix** MASKL:C attribute when set at some cells and not set at others, after editing the cell where it is set affected the other cells.
- Fixed: iup.**tabs** and iup.**cbox** were not allowing the creation of a control with no children in Lua.
- Fixed: secondary dialog for overwrite confirmation in **IupFileDlg** on Motif.
- Fixed: iup.**normalizer** when constructor receive children as parameters in Lua.
- Fixed: DIALOGFRAME attribute in **IupDialog** was handled only at map, affecting RESIZE processing before mapping.
- Fixed: SCROLLTO and SCROLLTOPOS in **IupText** on Windows.
- Fixed: ORIENTATION attribute in **IupSplit** were case sensitive.
- Fixed: DIRECTION attribute in **IupSbox** were case sensitive.
- Fixed: Enter key processing in **IupText** on Windows when MULTILINE=NO.
- Fixed: 'u' option processing (button names) of **IupGetParam** in Lua.
- Fixed: TIP attribute in **IupProgressBar** on Motif.
- Fixed: TIP attribute in **IupVal** on Windows.
- Fixed: TABTYPE was setting MULTILINE=NO when TOP or BOTTOM where set on Windows.
- Fixed: implemented missing iup.**GetChild** in Lua.
- Fixed: some images from **IupImageLibOpen** when using GTK.
- Fixed: invalid memory access in CURSOR attribute when name is too large.
- Fixed: BGCOLOR_CB and FGCOLOR_CB callbacks in Lua, where not properly cleaning the stack. (Thanks to zcs)
- Fixed: missing ih:**destroy**() method in Lua for some elements.
- Fixed: invalid memory access in X and Y attributes in **IupDialog** on the GTK driver when the dialog in hidden.
- Fixed: CARET attribute in **IupText** on Windows when the caret is located outside the visible area.
- Fixed: native destruction of **IupMenu** when inside a submenu on Windows.
- Fixed: image data end value in LEDC.
- Fixed: **IupNormalizer** parameter checking in LEDC.
- Fixed: invalid ampersand ('&') processing in TIPS on Windows. Improved ampersand processing on GTK.
- Fixed: VALUE=OFF display update in **IupToggle** on GTK when using an image in the toggle.
- Fixed: dialog layout now considers the global menu usage on the new Ubuntu Unity desktop.
- Fixed: SELECTION_CB in **IupTree** not being called for the last unselected node in a multi-selection tree, when that node is re-selected. on Windows that node was also not being selected.
- Fixed: on GTK changing focus was also changing the selection in a multiple selection **IupTree**.
- Fixed: invalid initialization of **IupList** when GTK version is older than 2.12.
- Fixed: redraw when FGCOLOR is set in **IupText** on Windows.
- Fixed: improved memory usage in variable parameter attribute functions.
- Fixed: MDI dialogs and menu behavior on Windows.

Version 3.5 (26/Apr/2011)

- New: attributes COUNT, LINECOUNT and LINEVALUE for **IupText**.
- New: dialog **IupElementPropertiesDialog** used internally at **IupLayoutDialog** now can be used by applications to inspect any element.
- New: common callback TIPS_CB called before a tooltip is displayed.
- New: CELLOFFSET*L*:*C* and CELLSIZE*L*:C attributes for **IupMatrix**.
- New: LIMITEXPAND attribute for **IupMatrix**.
- New: IUP_Webcam image in the IupImageLib.
- New: global attribute SHOWMENUIMAGES on GTK, with default value "Yes".
- New: NUMCOL_NOSCROLL and NUMLIN_NOSCROLL attributes for the **IupMatrix** that add more non scrollable cells.
- New: ORIGINOFFSET attribute for **IupMatrix**.
- **Changed:** TIPVISIBLE will now return the current visible state of the tip window.
- **Changed: IupConvertXYToPos** will now work for **IupMatrix** also.
- **Changed:** optimized redrawing of **IupCells** when SCROLLING_CB is not defined.
- **Changed:** removed "P" from **IupPPlot** additional API functions, because they will be used also for other plot controls. Old names still exists for compatibility.
- **Changed:** FRAMEVERTCOLOR*:C and FRAMEHORIZCOLORL:* are now also accepted in **IupMatrix**.
- Fixed: function iuplua_pushihandle when the element was not created in Lua, that cause a crash when destroying the Lua element.
- Fixed: destruction of the spin in a **IupText** element on Windows.
- Fixed: SELECTION and SELECTIONPOS attributes in **IupText** on Motif.
- Fixed: VALUE attribute returned in **IupFontDlg** on Motif.
- Fixed: DRAWSIZE attribute in **IupCanvas** on GTK when the canvas is hidden.
- Fixed: **IupColorBrowser** documentation was corrupted.
- Fixed: ACTION_CB callback not being called in **IupMatrix** when editing the cell and a non character key was pressed.
- Fixed: IUP_IGNORE is now processed in SPIN_CB in **IupText** on GTK.
- Fixed: VALUECHANGED_CB callback being called too many times in **IupText** on Windows and GTK.
- Fixed: the old BUTTON_RELEASE_CB callback in **IupVal** on GTK not being called.
- Fixed: multiline text size computation on Windows and Motif when the last line is empty.
- Fixed: VALUE* attributes in **IupTabs** on Motif when the new value is equal to the current value.
- Fixed: Ctrl+V, Ctrl+C, Ctrl+X and Ctrl+A key strokes were being inserted in the text in **IupText** on Motif.
- Fixed: CLIPBOARD attribute in **IupText** and **IupList** on Motif.
- Fixed: LEDC tool for **IupImageRGB** and **IupImageRGBA**.
- Fixed: TIP attribute in **IupTree** on Windows.
- Fixed: balloon tip attributes names to TIPBALLOON, TIPBALLOONTITLE and TIPBALLOONTITLEICON on Windows.
- Fixed: functions **IupPPlotInsertStrPoints** and **IupPPlotInsertPoints**.
- Fixed: invalid editing when using clipboard in **IupText** on Motif when READONLY=Yes.
- Fixed: invalid return value of READONLY attribute in **IupText** on Motif.

Version 3.4 (15/Feb/2011)

- New: function **IupClassMatch**.
- New: functions **IupPPlotInsertStrPoints**, **IupPPlotInsertPoints**, **IupPPlotAddPoints** and **IupPPlotAddStrPoints** for **IupPPlot** to add an array of samples at once.
- New: common attribute SCREENPOSITION that returs the X and Y attributes at once.
- New: ACTIVEWINDOW attribute for **IupDialog** on Windows and GTK.
- New: EDITNEXT attribute for **IupMatrix** to control the next cell after editing.
- New: FITTOTEXT action attribute, FITMAXHEIGHT and FITMAXWIDTH attributes in **IupMatrix**.
- New: INPUTCALLBACKS global attribute and GLOBALKEYPRESS_CB, GLOBALMOTION_CB, GLOBALBUTTON_CB and GLOBALWHEEL_CB global callbacks.
- New: **IupRecordInput** and **IupPlayInput** functions to record and play back mouse and keyboard interaction. (play partially working)
- **New:** VALUEMASKED attribute for **IupText**.
- **New:** SYSTEMLOCALE global attribute.
- Changed: removed compatibility with old bc55, gcc3 and mingw3 compilers on Windows. Depending on the Cygwin installation gcc3 may still works.
- Changed: improved internal class inheritance so iupClassNew can use iupRegisterFindClass to get its parent.
- Changed: NAVIGATE_CB callback in **IupWebBrowser** to process the return value. If IUP_IGNORE is returned navigation is canceled.
- Changed: improved compatibility with GTK 3.0.
- Changed: improved memory management in IupLua using the new LDESTROY_CB callback.
- Fixed: removed call to **cdCanvasFlush** when **IupPPlotPaintTo** is used.
- Fixed: FILTER and EXTFILTER attributes of **IupFileDlg** on GTK when more than one pattern is specified for the same filter ("*.jpg;*.bmp").
- Fixed: RESIZE=NO was incorrectly forcing MINBOX=NO for **IupDialog** on GTK.
- Fixed: compatibility with GTK 2.22.
- Fixed: padding warning on GTK when using **IupButton** with IMPRESS.
- Fixed: X and Y attributes in the GTK driver for all controls. X and Y attributes in the Windows and Motif drivers for the **IupDialog**.
- Fixed: **IupInsert** when adding the first element of a container.
- Fixed: set attribute in the properties dialog of the **IupLayoutDialog**.
- Fixed: behavior of BGCOLOR, GETFOCUS_CB, KILLFOCUS_CB, and K_ANY for **IupList** when DROPDOWN=Yes on GTK.
- Fixed: parameters "o" and "n" in iup.**GetParam** when used in Lua.
- Fixed: added support for **IupSplit**, **IupNormalizer** and **IupWebBrowser** in the ledc tool.
- Fixed: support for WIDTH*, HEIGHT*, RASTERWIDTH* and RASTERHEIGHT* attributes of **IupMatrix** when the **IupSetAttributeId** functions are used.
- Fixed: FITTOSIZE attribute in **IupMatrix** when title column size is defined by WIDTH0 or RASTERWIDTH0 to be 0.
- Fixed: WID get attribute in **IupLua** on Windows.
- Fixed: added missing iup.**GetNativeHandleImage** and iup.**GetImageNativeHandle** binding in IupLua.
- Fixed: line end converting when FORMATTING=Yes in **IupText** on Windows.
- Fixed: feedback when opening/closing branches in **IupTree** on Windows when SHOWDRAGDROP=YES.
- Fixed: DRAW_CB callback in **IupCells** when using the last parameter canvas.
- Fixed: cell value when editing was stared with a character not valid by the current MASK in **IupMatrix**.

Version 3.3 (release2) (18/Nov/2010)

We identified some limitations and problems with the new **IupWebBrowser** control, so we re-released some 3.3 packages to include an updated version of it.

- New: HTML attribute in **IupWebBrowser** to load a string. New COMPLETED_CB and ERROR_CB callbacks. New STATUS attribute.
- Changed: removed reason parameter from NAVIGATE_CB callback in **IupWebBrowser**. VALUE attribute will load always on the top frame.
- Fixed: VALUE attribute of **IupWebBrowser** was write only on Windows. Memory allocation that affected NAVIGATE_CB and NEWWINDOW_CB on Windows. Added missing Lua binding pre-compiled libraries.

Version 3.3 (09/Nov/2010)

- New: **IupWebBrowser** control using an embedded Internet Explorer on Windows, and Webkit in Linux.
- New: Perl binding for IUP by Kmx at GitHub.
- New: global attribute MOUSEBUTTON to send button press and button release messages.
- New: control **IupTuioClient** that connects to a TUIO server and process multi-touch messages.
- New: support for native multi-touch events in **IupCanvas** on Window 7. New TOUCH_CB and MULTITOUCH_CB callbacks. New TOUCH attribute.
- New: function **IupRefreshChildren** to update the layout locally at children only.
- New: IGNORE value for the FLOATING attribute.
- New: guide for building IUP, CD and IM in Linux on the documentation. Scripts for installation of the precompiled binaries or build binaries in the system.
- New: CLIENTOFFSET attribute for all containers.
- New: **IupLayoutDialog** pre-defined dialog to visually edit the layout of another dialog in run time.
- New: FLAT attribute for **IupToggle** when IMAGE is defined.
- New: AUTOREDRAW attribute for **IupTree** and **IupList** so redraw can be disabled on Windows.
- New: functions **IupCopyClassAttributes**, **IupGetClassCallbacks** and **IupGetAllClasses**.
- New: TABCHANGEPOS_CB callback in **IupTabs**.
- New: functions **Iup*AttributeId** to get and set attributes that need an ID. These functions are faster than the traditional functions because they do not need to parse the attribute name string and the application does not need to concatenate the attribute name with the id. The **IupMat*Attribute** functions also became faster than the traditional functions.
- New: parameters for **IupGetParam**, "o" to display the list in an array of toggles inside a radio, and "n" to select a font, similar to the "c" parameter that selects a color.
- New: DROPEQUALDRAG attribute for **IupTree**.
- New: **IupFontDlg** on Motif.
- New: FITTOSIZE action attribute in **IupMatrix**.
- New: callbacks BUTTON_CB, DROPFILES_CB, ENTERWINDOW_CB and LEAVEWINDOW_CB for **IupLabel**.

- New: the IupLua Console application now shows Lua code with syntax highlighting.
- Changed: MARQUEE attribute support in **IupProgressBar** on GTK and Motif now works just like on Windows.
- Changed: SHOWRENAME in **IupTree** can now be changed after map.
- Changed: Removed CLIENTSIZE1 and CLIENTSIZE2 from **IupSplit** and added CLIENTSIZE.
- Changed: TYPE attribute renamed to ORIENTATION in **IupVal** and **IupDial**. Old name still works.
- Changed: DIRECTION attribute renamed to ORIENTATION in **IupSplit**. Old name still works.
- Changed: removed FOCUSONCLICK from **IupButton**. The old name will set CANFOCUS.
- Changed: now when IMPRESS is defined along with IMAGE, and TITLE is not defined, then the borders will not be shown neither computed. The buttons with this attribute are now smaller than in previous versions.
- Changed: **IupReparent** to receive one more parameter to be used as a reference child.
- Changed: **IupSpinbox**, **IupSplit**, **IupSbox** now can be dynamically constructed with **IupAppend**/**IupInsert**.
- Changed: standard SIZE and RASTERSIZE format can also be used in **IupHbox**, **IupVbox** and **IupFill**.
- Changed: now **IupSaveClassAttributes** and **IupCopyClassAttributes** will save also id dependent attributes.
- Changed: Patch applied. Trying to improve the ADDFORMATTAG behavior in order to avoid scrolling physically and destroying the selection. Adds the concept of BULK format tags. The SELECTION and SELECTIONPOS attributes of the format tag will NOT change the **IupText** attributes anymore. Contribution of the Indentation library that adds syntax highlighting to a Lua code text. (Thanks to Nicolas N.)
- Changed: removed FINDUSERDATA attribute from **IupTree**, use **IupTreeGetId** always.
- Changed: removed images larger than 48x48 (inclusive) from the pre-compiled libraries of the IupImgLib, reducing its size and allowing more applications to use the pre-compiled binaries. The images are still available in the source code.
- Fixed: callback not called in **IupGetParam** when a file name or a color string are selected by the extra dialog button.
- Fixed: LEDC to correctly include the necessary headers.
- Fixed: FRAMEVERTCOLORL:C attribute of **IupMatrix** for cell with col=0 or lin=0.
- Fixed: avoid double calls to ACTION in **IupButton** on Windows when pressing enter and a dialog is displayed.
- Fixed: the cleaner syntax feature for separator creation in Lua.
- Fixed: returned value in RASTERSIZE for the **IupDialog** on GTK and Motif.
- Fixed: missing **IupSetCallbacks** export for "iup.dll".
- Fixed: **IupFileDlg** on Motif when MULTIPLEFILES=Yes and FILE_CB is not defined.
- Fixed: natural height computation for **IupList** on Windows when DROPDOWN=Yes.
- Fixed: ACTION callback called an extra time when FOCUSONCLICK=NO and user double click in **IupButton** on Windows.
- Fixed: TABTITLEn attribute return value of **IupTabs** when TABTITLE was set at a child.
- Fixed: SCREENSIZE global attribute on GTK when using multiple monitors.
- Fixed: MARKEDid attribute in **IupTree** on Windows when MARKMODE=SINGLE, was not changing the focus node.
- Fixed: line end converting when FORMATTING=Yes in **IupText** on Windows. (Thanks to Nicolas N.)
- Fixed: **IupMessageDlg** modality on Windows, when PARENTDLG is not specified then it will be modal relative to all application dialogs.
- Fixed: mnemonic support for TABTITLE on GTK and Motif.
- Fixed: drag&drop, move and copy of nodes in **IupTree** on GTK.
- Fixed: mapping of standard font names to Pango names on GTK.
- Fixed: invalid current cell in **IupMatrix** after adding lines or column to a matrix that had 0 lines or 0 columns.
- Fixed: **IupSetFocus** was not working on GTK if the dialog does not has the focus.
- Fixed: RESIZE_CB callback in **IupCanvas** not being called after the canvas is mapped on GTK.
- Fixed: removed Scrollbar warning when creating a multiline **IupText** on Motif.
- Fixed: FONT handling in **IupText** on Windows when FORMATTING=Yes.
- Fixed: Enter key processing when editing a **IupMatrix** cell and IUP_IGNORE is returned in EDITION_CB, to avoid propagating that key press to the dialog.
- Fixed: **IupZbox** VALUE management when the zbox itself is not visible.
- Fixed: **IupSplit** when AUTOHIDE=Yes.
- Fixed: an invalid focus cell in **IupMatrix** could occur after NUMLIN or NUMCOL was changed to smaller values.
- Fixed: invalid call to ACTION callback of a **IupToggle** when inside a radio and VALUE is set.

Version 3.2 (26/June/2010)

- New: function **IupResetAttribute**.
- New: MINMAX attribute for **IupSplit**.
- New: global attribute SINGLEINSTANCE so the application can restrict the number of its instances on Windows. COPYDATA_CB callback for **IupDialog** on Windows to receive the command line of the secondary instances when SINGLEINSTANCE is used.
- New: attributes WMFAVAILABLE, EMFAVAILABLE, SAVEWMF and SAVEEMF for **IupClipboard** on Windows.
- Changed: some Lua parameters that use "number" to "integer".
- Changed: replaced old "arg" usage for "..." to improve better compatibility with LuaJIT. (Thanks to J.-F. Cap.)
- Changed: **IupSaveClassAttributes** to not save values that are equal to the default values.
- Changed: **IupFileDlg** behavior on Motif to avoid auto resize when a directory is changed.
- Changed: C function **iuplua_dofile** will now check for the IUPLUA_DIR environment variable when file could not be opened.
- Changed: control of LOHs inclusion moved from the source code to the makefile.
- Changed: removed compatibility with require"iuplua51", now LuaBinaries must be used or LUA_CPATH must be set.
- Changed: added compatibility with Lua 5.2.
- Changed: global variable SYSTEM in Mac OS X, renamed from Darwin to MacOS.
- Fixed: the cleaner syntax feature for menu creation in Lua.
- Fixed: stack overflow when "MARKL:C" is set inside MARKEDIT_CB because MARK_CB is not defined in **IupMatrix**.
- Fixed: **iup.TreeSetUserId** error in Lua.
- Fixed: **IupView** executable in Win64.
- Fixed: RIGHTCLICK_CB called twice when **IupTree** is inside two **IupTabs** on Windows.
- Fixed: **IupLoopStep** on Windows to process the IDLE callback.
- Fixed: natural size of **IupText** and **IupMatrix** when SCROLLBAR is VERTICAL or HORIZONTAL only.
- Fixed: **IupSplit** were not considering MINSIZE and MAXSIZE.
- Fixed: EXPANDWEIGHT implementation.
- Fixed: MARK=CLEARALL in **IupTree** on Windows. DELNODE=MARKED in **IupTree** on all drivers, so the node 0 can also be removed.
- Fixed: return value of SIZE and RASTERSIZE of the **IupDialog** when reset to NULL after the dialog was mapped.
- Fixed: a right click in a node in **IupTree** on Windows was causing an invalid redraw of the selected node.
- Fixed: parameter indexing when using the new button names parameter in **IupGetParam**.

Version 3.1 (22/Apr/2010)

- New: MULTIUNSELECTION_CB callback in **IupTree**. MARKEDNODES attribute in **IupTree**.
- New: EXPANDWEIGHT attribute for children of **IupVbox** and **IupHbox**.
- New: HIDDENTEXTMARKS attribute in **IupMatrix**. ALIGNMENT attribute for all cells in **IupMatrix**, used when ALIGNMENTc is not defined.
- New: **IupSplit** control, similar to **IupSbox** but with two children.
- New: option "u" in **IupGetParam** to redefine the OK and Cancel buttons names and add a Help button.
- New: ADDROOT attribute in **IupTree**, its default is Yes. The first node now can be deleted and can have other nodes at depth=0. New DELNODE=ALL value that remove all nodes.
- New: native driver for MacOSX under construction. Help needed! (Thanks to Heesob P.)
- Changed: INSERTITEMn attribute in **IupList** now allows to add an item after the last item using n=count+1.
- Changed: removed **IupMessage** from error handling in IUP-IM utilities. Now a global attribute "IUPIM_LASTERROR" is set instead.
- Changed: NUMCOL_VISIBLE and NUMLIN_VISIBLE attributes can now be greater than the actual number of columns/lines, so room will be reserved for adding new columns/lines without the need to resize the matrix.
- **Changed: IupTree** internal optimization for **IupTreeSetUserId**, **IupTreeGetId** and **IupTreeGetUserId**.
- **Changed:** removed AUTODRAGDROP attribute from **IupTree** on GTK.
- **Changed:** added support for multiple file selection in **IupFileDlg** on Motif.
- **Changed:** Simplify IupLua implementation. More similar to a regular binding implementation like CDLua and IMLua.
- **Changed:** removed old controls **IupTabs** and **IupVal** kept for compatibility in the IupControls libraries. The new **IupTabs** and **IupVal** in the main library must be used from now on.
- Fixed: optional creation parameter of **IupSubmenu**, **IupSbox**, **IupFrame**, **IupRadio**, **IupVal** and **IupDial** in Lua.
- Fixed: **IupSbox** handler feedback when MAXSIZE or MINSIZE is used.
- Fixed: color value in **IupGetParam** after the color browser dialog canceled editing.
- Fixed: improved compatibility in **IupOleControl** so it can be compiled with gcc from Cygwin.
- Fixed: display update when changing IMAGE attribute in **IupLabel** on Windows. Display update when changing FGCOLOR and ALIGNMENT attributes in **IupToggle** on Windows.
- Fixed: default image update in **IupTree** on Motif.
- Fixed: scrollbar position in **IupCanvas** on Windows after dragging the thumb when it is released.
- Fixed: NUMCOL_VISIBLE and NUMLIN_VISIBLE default value in **IupMatrix**.

- Fixed: **IupMatrix** scrolling can now position cells in intermediate positions. When using the scrollbar button still scrolls in cell steps, when dragging the scrollbar can freely position the cells. This fixes the problem of the last column or line being incomplete visible.
- Fixed: vertical frame drawing of a cell when using FRAMEVERTCOLORL:C equals to the background color. FRAMEVERTCOLORL:C and FRAMEHORIZCOLORL:C can now use "BGCOLOR" as value to not draw the frame line.
- Fixed: added missing exports in the main DLL for **IupGetInt2** and **IupGetIntInt**.
- Fixed: object position in **IupOleControl**. New sample using LuaCOM for callbacks. (Thanks to Kommit)
- Fixed: background color for images in **IupTabs** and **IupTree** when not using visual styles on Windows.
- Fixed: integer input mask when limited to min-max in **IupGetParam**.
- Fixed: invalid memory access on multiple selection callback management in **IupList**.
- Fixed: compatibility code for deprecated functions on GTK 2.20.
- Fixed: FILE_CB callback file parameter in **IupFileDlg** when multiple files are selected on Windows.
- Fixed: RENAME_CB callback being called when user cancel renaming in **IupTree** on Windows. BRANCHOPEN_CB or BRANCHCLOSE_CB being called when STATEid is set in **IupTree** on GTK and on Motif.
- Fixed: invalid memory access when saving DIRECTORY in a multiple selection **IupFileDlg** on GTK.
- Fixed: dropdown cell editing in **IupMatrix** on Motif.
- Fixed: invalid memory access in hash table module when removing an attribute.

Version 3.0.1 (14/Mar/2010) (Internal)

- New: TOTALCHILDCOUNTid and LASTADDNODE attributes in **IupTree,** so you can compute or retrieve the id of the node created by an INSERT operation.
- New: common callback DESTROY_CB.
- Changed: propagation of attributes will be ignored at a child where the attribute is marked as non inheritable.
- Changed: double click sequence of events on GTK to match the sequences on Windows and Motif.
- Changed: when IMAGE is defined for a **IupButton**, TITLE will be also considered during map if not NULL and not empty. This will allow buttons with images created in LED to continue to use "" to define their titles.
- **Changed:** When the DIRECTORY attribute of **IupFileDlg** is consulted after the dialog is closed and the user pressed the OK button, it will contain the directory of the selected file.
- **Changed: IupTree** internal optimization to match ids to/from native nodes. SHOWRENAME_CB callback return code to accept IUP_IGNORE. The NODEREMOVED_CB callback now only receive the node userdata.
- Fixed: **IupGetParam** param parsing of %f and %c in Lua.
- Fixed: **IupTreeUtil** contributed functions.
- Fixed: AXS_YREVERSE attribute in **IupPPlot**.
- Fixed: size of a node in **IupTree** on Windows when TITLEFONT is set.
- Fixed: LEGENDPOS attribute in IupPPlot.
- Fixed: invalid call to SELECTION_CB callback in **IupTree** on Windows when MARKMODE=MULTIPLE and the user pressed the Ctrl key to select an item. Missing call to SELECTION_CB on GTK and Motif when user unselect multiple nodes at once. Invalid change of the selection when focus is changed in **IupTree** on Windows Vista/7 when not using visual styles. on GTK and Motif children of not expanded nodes were not being selected when multiple nodes were selected in one operation.
- Fixed: interpretation of comments inside elements declaration in LED.
- Fixed: invalid memory access in **IupClose** when using LED.
- Fixed: selection was not hiding in **IupText** when the control loses its focus on Windows when MULTILINE=NO.
- Fixed: MARKMODE in **IupTree**, NC in **IupText** and **IupList**, PADDING in **IupLabel, IupButton, IupToggle, IupText** and **IupTabs**, if they were set only before map.

Version 3.0 (26/Jan/2010)

- New: added support for MacOSX using GTK.
- New: function **IupLoopStepWait**.
- New: functions **iup.TreeSetNodeAttributes**, **iup.TreeSetAncestorsAttributes** and **iup.TreeSetDescentsAttributes** for IupTree in Lua. (Thanks to Tomas G.)
- Changed: **iup.TreeSetValue** renamed to **iup.TreeAddNodes**. Old name also works.
- Fixed: **IupList** with DROPDOWN=Yes and the last item is removed.
- Fixed: dynamic BGCOLOR change on Windows for **IupText**, **IupList**, **IupVal** and **IupTabs**.
- Fixed: duplicate call to TABCHANGE_CB in **IupTabs** on Windows when a Tab is inside another Tab.
- Fixed: IUP_IGNORE support in **IupText** on Windows for the K_ANY callback.
- Fixed: focus management when dialog gets back the focus and must redirect it to the last child with focus, on Windows.
- Fixed: button press feedback when FOCUSONCLICK=NO in **IupButton** on Windows XP.
- Fixed: title bar display in **IupDialog** on GTK when only TITLE=NULL, but MENUBOX is still YES.
- Fixed: default value for VALUE in **IupFontDlg**.
- Fixed: background color of edit box in **IupTree** on Windows when not using Visual Styles.
- Fixed: CARET attribute in **IupText** on Windows when line is greater than the last line.
- Fixed: excess of motion_cb events in **IupCanvas** on GTK when in UNIX.
- Fixed: CMARGIN attribute in **IupVbox** and **IupHbox**.
- Fixed: invalid memory access in NODEREMOVED_CB callback processing of **IupTree** on Windows.
- Fixed: VALUE attribute in **IupTree** when MARKMODE=SINGLE, on Windows was not unselecting the previous node, on GTK if set during the SELECTION_CB was aborting the next call to the callback. on GTK and Motif was also not showing the node if inside a collapsed branch.

Version 3.0 RC 4a (18/Dec/2009)

- Fixed: VISIBLE attribute management. **IupZbox** now will respect if a child has a VISIBLE attribute set, and it will not change it. **IupTabs** now does not depends on the VISIBLE attribute anymore.
- Fixed: VALUE attribute return in **IupItem** on GTK.

Version 3.0 RC 4 (14/Dec/2009)

- New: NMARGIN and NGAP non-inheritable attributes for **IupHbox** and **IupVbox**.
- New: "OTHER" status code for FILE_CB when selecting an invalid file name or a directory in **IupFileDlg**.
- New: DLL_HINSTANCE global attribute on Windows.
- Changed: Added a workaround for TITLEFONTid for **IupTree** when changing only the Bold style on Windows.
- Changed: the RENAMENODE_CB callback in **IupTree** is not supported anymore.
- Changed: improved compatibility of **IupFileDlg** when DIALOGTYPE=DIR and CoInitializeEx was initialized with COINIT_MULTITHREADED prior to **IupOpen** on Windows.
- Changed: **IupFrame** can now has a color background when not using TITLE, and BGCOLOR is set before map.
- Fixed: memory leak in **IupPPlot**.
- Fixed: invalid memory access in set ALIGNMENT attribute for **IupLabel**, **IupButton** and **IupToggle**, and in set MARK for **IupTree**.
- Fixed: invalid layout computation when using the old **IupSpin** element.
- Fixed: STATE attribute for **IupTree** on Windows when branch has no child.
- Fixed: invalid redraw of some controls when dialog is resized on Windows.
- Fixed: invalid memory access for SYSTEMVERSION global attribute in Linux. (Thanks to David G.)
- Fixed: missing convertion to UTF-8 in **IupButton** when handling TITLE at map in the GTK driver.
- Fixed: image branch update when branch STATE is changed in **IupTree** on Windows.
- Fixed: SHOWRENAME_CB callback when renaming is started clicking twice in **IupTree**.
- Fixed: invalid limit check in VALUE attribute of **IupList** in the GTK driver. (Thanks to Paul G.)
- Fixed: invalid memory access when setting VALUE to NULL in **IupTree**.
- Fixed: ACTION callback called when an item is set on a **IupList** when DROPDOWN=Yes.
- Fixed: dialog decoration size when menu is associated during the map process.
- Fixed: K_ANY callback called twice for **IupTabs** in the GTK driver.
- Fixed: invalid memory access when destroying some of the additional controls that use CD.
- Fixed: incomplete redraw of the **IupCanvas** on Windows XP when a window moves over the canvas.
- Fixed: missing call to ACTION when an item that was replaced is clicked in **IupList**.
- Fixed: switch of a complete menu in **IupDialog** was not working.
- Fixed: button press feedback when FOCUSONCLICK=NO in **IupButton** on Windows.
- Fixed: VISIBLE attribute for non native containers. It affected **IupZbox**.
- Fixed: **IupMatrix** with EXPAND=NO was behaving as EXPAND=YES.

Version 3.0 RC 3 (02/Oct/2009)

- New: MOVE_CB callback for **IupDialog** on Windows and GTK.

- New: SPINNING attribute for **IupGetParam** when the callback is activated by a spin.
- New: KEYPRESS, KEYRELEASE and KEY global attributes.
- New: MAXSIZE and MINSIZE attributes for all controls.
- New: NODEREMOVED_CB callback for **IupTree**.
- New: SORT attribute for **IupList**.
- New: function **IupSaveImageAsText**.
- New: function **IupLoadBuffer**.
- New: parameter in the EDITION_CB callback of **IupMatrix** to indicate if the value will be updated.
- New: auxiliary functions **IupGLUseFont** and **IupGLWait** for the **IupGLCanvas**. attribute REFRESHCONTEXT on Windows.
- **New:** VALUECHANGED_CB callback for **IupVal**, **IupDial**, **IupColorBrowser**, **IupToggle**, **IupText** and **IupList**.
- **New:** element **IupClipboard**.
- **New:** functions **IupGetNativeHandleImage** and **IupGetImageNativeHandle** for the Iup-IM library.
- Changed: now the **iup.image** constructor also accepts parameters in the same format as **iup.imagergb** and **iup.imagergba**.
- Changed: return value to boolean of **iup.GLIsCurrent, iup.GetParam, iup.SaveImage, iup.isshift, iup.iscontrol, iup.isbutton1, iup.isbutton2, iup.isbutton3, iup.isbutton4, iup.isbutton5, iup.isdouble, iup.issys, iup.isalt, iup.isSysXkey, iup.isAltXkey, iup.isCtrlXkey, iup.isShiftXkey** and **iup.isXkey** in Lua.
- Changed: the function **iup.key_open** is now obsolete and not necessary anymore.
- Changed: improved transparency for 8bpp images on Windows.
- Changed: in **IupMatrix** since the selection is made only using the mouse, by pressing a key will NOT clear the selection anymore. You can still do that setting MARKED=NULL in the K_ANY callback. Improved MARKL:C to be more flexible for other MARKMODE options.
- Changed: updated the **IupTreeUtil** contributed utility.
- Changed: CHANGEVALUE_CB callback renamed to VALUECHANGED_CB in **IupVal**.
- Changed: internal reorganization of the abstract layout methods of the Ihandle class to allow more flexibility and control of the layout process.
- Changed: LAYERED and LAYERALPHA attributes are now condensed in the OPACITY attribute. The OPACITY is available on Windows and GTK.
- Fixed: the functions **IupPreviousField** and **IupNextField** to respect the dialog hierarchy order.
- Fixed: NUMCOL and NUMLIN when set to 0 in **IupMatrix**. Double click in a title cell was entering in edit mode at the focus cell. Marks were processed after ENTERCELL_CB when the user single click a cell. Enter key processed also for the next cell when MULTIPLE=YES after editing ended.
- Fixed: STARTFOCUS on Motif and Win32 for **IupDialog** where not working. Now STARTFOCUS is set only if SHOW_CB did not changed the current focus.
- Fixed: DLGBGCOLOR on Motif where incorrectly set.
- Fixed: **IupToggle** redraw inside an **IupFrame** on Windows XP where disapearing.
- Fixed: background color of the edit box of **IupTree** on Windows XP where black.
- Fixed: release of stock images in **IupClose** caused the application to crash.
- Fixed: auxiliar function **iup.TreeSetUserId** in Lua when releasing the previous reference.
- Fixed: ACTION callback of **IupButton** on Windows when FOCUSONCLICK=NO was not being called.
- Fixed: return value of **IupSaveImage** was inverted.
- Fixed: export of image in Lua at the **IupView** application.
- Fixed: **IupGetParam** when specifying full intervals without the step parameter.
- Fixed: DEFAULTENTER and DEFAULTESC on Windows when focus is inside an **IupTabs**. Also on Windows they were processed before K_ANY, so K_ANY could not abort them by returning IUP_IGNORE.
- Fixed: K_ANY called twice for K_CR when **IupText** has multiple lines on Windows.
- Fixed: on Windows when a pre-defined system dialog was closed with Enter or Esc, the key was propagated to the dialog that open it.
- Fixed: keyboard navigation in the dialog now respects the order of **IupNextField** and **IupPreviousField** for all drivers. Those functions were also improved.
- Fixed: on GTK the VISIBLE attribute returned invalid result when child is hidden by its parent.
- Fixed: on Windows the text color of a selected item of an **IupTree** was not inverted.
- Fixed: on Windows the VALUE attribute of a inactive **IupItem** was always OFF.
- Fixed: ENTERWINDOW_CB and LEAVEWINDOW_CB for **IupCanvas** on Windows were not being called.
- Fixed: HELP_CB was not working for **IupVal**, **IupTabs** and **IupTree** on Motif.
- Fixed: USETITLESIZE attribute logic in **IupMatrix**.
- Fixed: DELNODE attribute when value is CHILDREN in **IupTree**. It was not working for the root node.

Version 3.0 RC 2 (18/Jul/2009)

- New: MONITORSINFO and VIRTUALSCREEN global attributes now also available on GTK.
- New: USETITLESIZE attribute for **IupMatrix**.
- New: DEFAULTFONTSIZE global attribute.
- New: **IupSetAtt** auxiliar function.
- Changed: the default alignment for **IupButton** (Text and Image) and IupToggle (Image) to "ACENTER:ACENTER".
- Changed: improved decoration size computation for **IupDialog** on GTK.
- Fixed: **IupItem** on GTK when compiled in versions older than 2.14, but run in newer versions.
- **Fixed:** alignment of buttons in **IupAlarm**.
- **Fixed: IupZbox** visible child management and VISILBE attribute update after mapping an element.
- **Fixed:** X and Y attributes for GTK.
- **Fixed: IupTree** TITLE with non UTF-8 characters.
- **Fixed: IupClose** in loop when removing names.
- **Fixed:** CONTEXT and VISUAL in **IupGLCanvas**.
- **Fixed:** SHOWTICKS in **IupVal**.
- **Fixed:** in **IupMatrix.** default cell alignment. BGCOLOR and FGCOLOR to use the global default colors instead of "255 255 255" and "0 0 0". drawing details. misbehavior of the scrollbar on GTK. improved IUP 2 compatibility when calling VALUE_CB and when consulting titles to compute cell size.
- **Fixed:** VALUE management in **IupZbox**.
- **Fixed:** removed "cannot add non scrollable widget" warning message when creating a **IupCanvas** on GTK.
- **Fixed:** ADDEXPANDED in **IupTree**.
- **Fixed:** SIZE consideration in layout computation for **IupDialog**.
- **Fixed:** DIALOGTYPE=MESSAGE for **IupMessageDlg** on GTK.
- **Fixed: IupButton** with no text and no image, but with BGCOLOR defined will properly show the color.

Version 3.0 RC 1 (26/Jun/2009)

General

- New: checked for memory leaks using VLD on Windows and Valgrind in Linux.
- New: PREVIEWGLCANVAS attribute for **IupFileDlg**.
- New: auxiliary functions **IupTextConvertLinColToPos** and **IupTextConvertPosToLinCol** for **IupText**.
- New: basic tutorial for IupLua. (Thanks to Steve D.)
- New: **IupTree** now uses native controls and was moved to the standard controls. The old implementation is not available. Images for nodes are not limited to 16x16 anymore. BGCOLOR now follows the same default as **IupText** and **IupList**, and can be changed. New TITLEFONT, FGCOLOR, USERDATA, FINDUSERID, COUNT, CHILDCOUNT, EXPANDALL, INDENTATION, HIDEBUTTONS, HIDELINES, COPYNODE, MOVENODE, SPACING, TOPITEM, INSERTLEAF and INSERTBRANCH attributes. New BUTTON_CB, MOTION_CB and DROPFILES_CB callbacks. Attributes SCROLLBAR and REDRAW are not supported anymore. VALUE attribute split in VALUE and MARK attributes, set MARK using VALUE is still possible fro backward compatibility. STARTING renamed to MARKSTART, and CTRL/SHIFT attributes replaced by MARKMODE (old names kept working for compatibility). Now if DRAGDROP_CB returns IUP_CONTINUE or if it is not defined but SHOWDRAGDROP=Yes then the node will be automatically moved to the new position. **ATTENTION -** DEPTH is now a read-only attribute, use the INSERT* attributes to properly add nodes. NAMEid attribute renamed to TITLE, old attribute still works but will be removed in future versions since it conflicts with the common NAME attribute. The SELECTION_CB and MULTISELECTION_CB callbacks now ignore their return value. The rename action is now activated by two clicks instead of a double click.
- **Changed**: removed "lua5.1.so" dependency in UNIX.
- Changed: In IupLua the Lua function **iup.TreeSetValue** now also accepts node decoration in the initialization table and can add a subtree to any node. (Thanks to Tomas G.)
- Changed: In IupLua attributes that are pointers to Ihandle are now returned as ihandle instead of userdata.
- Changed: replaced "[]" in function declarations by a simple "*". None of those functions needed it.
- Changed: the default value of the Windows attribute COMPOSITED is back to NO to improve backward compatibility and to avoid side effects of the attribute.
- Changed: the auxiliary functions **IupTextConvertXYToChar** and **IupListConvertXYToItem** where replaced by **IupConvertXYToPos**, that also works for **IupTree**.
- Changed: added support for WHEEL_CB on GTK for **IupCanvas**.
- Fixed: IupLua initialization when retrieving the argc/argv arguments for **IupOpen**. (Thanks to Ross B.)
- Fixed: Arg initialization for all controls on Motif driver.
- Fixed: update of the POSX and POSY attributes for the **IupCanvas**.
- Fixed: FONT size round when converting from pixels to points on Windows. (Thanks to Devin S.)
- Fixed: button disappearing after mouse over on Windows XP.
- Fixed: **IupMatrix** when NUMCOL/NUMLIN were less than NUMCOL_VISIBLE/NUMLIN_VISIBLE. Also fixed when NUMCOL/NUMLIN were 0 and changed to 1, and when removed 1.

CURSOR attribute when RESIZEMATRIX=Yes. (Thanks to Jeremy C.)
- Fixed: action callback return value in Lua for the **IupGetParam** dialog. (Thanks to Zhiwei)
- Fixed: EXPAND attribute for **IupCanvas**.

Version 3.0 BETA 3 (04/Apr/2009)

- New: MARKL:C, READONLY, NUMLIN_VISIBLE_LAST, NUMCOL_VISIBLE_LAST, and SHOW attributes for **IupMatrix**. When scrolling the matrix using the scrollbar the focus is not changed anymore. The last cells at right and bottom are now drawn as incomplete cells if they do not fit in the visible area. New FONT_CB callback. CHECKFRAMECOLOR is not necessary anymore, just set FRAMEVERTCOLOR or FRAMEHORIZCOLOR. Internal code reorganization. AREA and MULTIPLE renamed to MARKAREA and MARKMULTIPLE, old names as still supported. New MULTILINE attribute to edit text in multiple lines, valid only before mapped.
- New: **IupRedraw** and **IupSetClassDefaultAttribute** functions.
- Changed: Added package registration code to IupLua that allows it to be statically linked and require"iuplua" does not abort if the iuplua_open function was called.
- Changed: the **IupOleControl** in Lua will not automatically initialize LuaCOM anymore. The application must manually call "elem:CreateLuaCOM()". The previous initialization was incorrect (thanks to Ross B.).
- Changed: the declaration of function **IupGetClassAttributes** to use the class name instead of a control handle.
- Fixed: Fixed button, toggle and list sizes for GTK driver when using the Hildon Framework. (Thanks to Otfried C.)
- Fixed: some IupLua dynamic libraries in Linux where incorrectly linking with Motif (libiuplua + pplot, cd, controls, gl, im and imglib + 51.so)
- Fixed: HOMOGENEOUS attribute for **IupVbox** and **IupHbox**.
- Fixed: CARET attribute on GTK driver was not correctly scrolling the multiline text when not visible.
- Fixed: parameter checking and the return value in Lua for **IupListDialog** when type=2.
- Fixed: the return value for **IupGetText** when the user canceled. (Thanks to Xu W.)
- Fixed: **IupGetClassAttributes** and **IupGetAllAttributes** were not implemented in IupLua.
- Fixed: The 32 bits version of the IupLua console on Windows XP64 was not working.
- Fixed: CARET_CB and **IupTextConvertXYToChar** in **IupText** when MULTILINE=YES and FORMATTING=NO.

Version 3.0 BETA 2 (26/Dec/2008)

- Changed: **ATTENTION -** the following headers were deprecated iupcb.h, iupcells.h, iupcolorbar.h, iupdial.h, iupgauge.h, iupmatrix.h, iuptree.h - use iupcontrols.h only
- Changed: **ATTENTION -** the following headers were deprecated iupgetparam.h, iupspin.h, iuptabs.h, iupval.h - use iup.h only
- Fixed: set VALUE attribute for IupText on Windows when formatting is used.
- Fixed: **IupHide** when dialog was maximized on Windows.
- Fixed: get VALUE attribute for **IupText** in all drivers, after the element is mapped it must return the empty string "" when there is no text.
- Fixed: **IupGetParam** when specifying partial intervals.
- Fixed: K_Esc key callback processing on Windows.
- Fixed: PLACEMENT and FULLSCREEN for IupPopup.

Version 3.0 BETA 1 (15/Dec/2008)

General

- New: GTK driver, available in UNIX and Windows.
- New: internal code reorganization. More clear and simple to create controls and drivers. All comments are now in English.
- New: internal documentation and Guide to create new controls. Now all the controls use the same architecture using the same base class.
- New: IUP_ASSERT compile flag.
- New: **IupMainLoopLevel** function.
- New: support for the HILDON framework that runs on top of GTK on the Maemo platform used by the Nokia Internet Tablets. (Thanks to Otfried C.)
- Changed: all dialogs, and all elements that have names, are now automatically destroyed in **IupClose**.
- Changed: **ATTENTION -** the following headers were deprecated iupcbox.h, iupsbox.h - use iup.h only
- Changed: **ATTENTION -** the headers iupcompat.h and iupcpi.h were removed. They are not supported anymore.

Common Attributes

- New: CHARSIZE convertion factor used by the SIZE attribute.
- New: NAME used by **IupGetDialogChild.**
- New: font face name mappings for Courier, Times and Helvetica.
- New: functions **IupGetClassAttributes, IupGetIntInt**.
- New: CLIENTSIZE returns the size of containers excluding their decoration.
- New: TIP additional attributes (Motif and Windows): TIPFONT, TIPDELAY, TIPBGCOLOR, TIPFGCOLOR, TIPBALLON (Windows Only), TIPBALLONTITLE (Windows Only), TIPBALLONTITLEICON (Windows Only), TIPVISIBLE. Not available on GTK.
- New: TIPRECT auxiliary attribute for the TIP common attribute.
- Changed: attribute FONT now uses a common a more flexible definition for all drivers, old format is still supported. The default FONT on Motif is now "Fixed, 10".
- Changed: **ATTENTION -** Now attributes are stored in the internal hash table only if not processed or allowed by the element class implementation.
- Changed: **IupGetAttribute**, **IupSetAttribute** and **IupStoreAttribute** can also be used to access global attributes using NULL as element.
- Changed: TIP and ZORDER attributes are now non inheritable.
- Changed: **ATTENTION -** the BGCOLOR is now ignored in **IupLabel**, **IupFrame**, **IupToggle** (for the text) and **IupVal**. They will use the background color of the native parent.

Global Attributes

- New: APPSHELL, XDISPLAY, XSCREEN, XSERVERVENDOR, XVENDORRELEASE on Motif.
- New: VIRTUALSCREEN and MONITORSINFO on Windows.
- Changed: LANGUAGE default from PORTUGUESE to ENGLISH.
- Changed: TRUECOLORCANVAS and SYSTEMLANGUAGE are now available in all drivers.

Common Callbacks

- New: IUP_IGNORE return code accepted for **IDLE_ACTION** callback to automatically remove the callback.
- New: UNMAP_CB for all controls
- Changed: MAP_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB for all controls.

Layout

- New: functions **IupGetDialogChild, IupUnmap, IupReparent, IupInsert, IupUpdateChildren**, **IupGetClassType, IupGetChildPos** and **IupGetChildCount.**.
- New: FLOATING attribute to control the inclusion of the element in layout processing for **IupHbox**, **IupVbox** and **IupZbox**.
- New: HOMOGENEOUS attribute to control the spacing in layout processing for **IupHbox** and **IupVbox**.
- New: EXPANDCHILDREN attribute to control the expansion in layout processing for **IupHbox** and **IupVbox**.
- New: NORMALIZESIZE attribute to control the natural size in layout processing for **IupHbox** and **IupVbox**.
- New: element **IupNormalize**r.
- New: CGAP and CMARGIN for **IupVbox** and **IupHbox** that use SIZE units.
- New: VALUEPOS and VALUE_HANDLE attributes for **IupZbox**.
- Changed: default value for ALIGNMENT in **IupZbox** is now "NW".
- Changed: **IupAppend** and **IupDetach** can now be used for dynamic creation of menus or containers, even after the element is mapped.
- Changed: **IupDetach** will now automatically unmap the element.
- Changed: **IupAppend** will now return the actual parent.
- Changed: **IupUpdate** now only mark the control to be redraw instead of redrawing at the function call.

Dialogs

- New: MINSIZE and MAXSIZE attributes. on Windows MINSIZE is ignored for systems with multiple monitors. The Windowing system may impose a minimum default limit for the dialog that includes the title bar with all it buttons.
- New: DROPFILES_CB and RESIZE_CB callbacks.
- New: IUP_CURRENT and IUP_CENTERPARENT positions for **IupShowXY** and **IupPopup**.

- New: IUP_HIDE and IUP_MAXIMIZE flags for SHOW_CB callback.
- New: MODAL attribute to check if the dialog was shown with **IupShow** or **IupPopup**.
- New: **IupColorDlg, IupFontDlg** and **IupMessageDlg** native pre-defined dialog as elements.
- New: SHOWHIDDEN attribute for **IupFileDLg**. Preview canvas support for the Motif driver.
- New: tip string for each param in **IupGetParam**. And a new "c" param to show a RGB color string with extra controls to show the color and open the color selection dialog.
- Changed: SAVEUNDER dialog attribute now is also available on Motif.
- Changed: DROPFILES_CB callback is now available for all controls. It is only activated using DRAGDROP attribute. It is active by default only for **IupCanvas** and **IupDialog**.
- Changed: the default value of the Windows attribute COMPOSITED is now YES, except on Windows Vista.
- Changed: **IupDestroy** is now automatically called for child dialogs when the parent is destroyed.

Canvas

- New: LINEX, LINEY, XAUTOHIDE and YAUTOHIDE attributes for the scrollbar.
- New: CLIPRECT attribute, a rectangle that has its region invalidated for paiting.
- Changed: if ACTION is defined nothing is painted in the canvas, now also on Motif.
- Changed: BORDER is now also supported on Motif.
- Changed: **ATTENTION -** now scrollbar parameters min, max, page size and line size are updated when DX/DY are updated. POSX and POSY will only update the position of the scrollbar. Automatic hide of the scrollbar now works also on Motif.

Label, Button and Toggle

- New: attributes PADDING, ELLIPSIS, WORDWRAP and MARKUP for **IupLabel**.
- New: IMPRESSBORDER, PADDING, MARKUP, FOCUSONCLICK and ALIGNMENT attributes for **IupButton**.
- New: support for image and text simultaneous in **IupButton**.
- New: support for mnemonics in **IupLabel**, **IupButton** and **IupToggle**.
- New: RADIO attribute for **IupToggle**.
- Changed: ALIGNMENT attribute now includes vertical alignment values.
- Changed: **IupButton** now supports text with more than one line.

Text and Multiline

- New: APPENDNEWLINE and PADDING attributes. CUEBANNER and FILTER attributes on Windows.
- New: MASK attribute for **IupText**, **IupMultiline**, **IupList** and **IupMatrix**. The iupmask functions are now obsolete, autofill option and MATCH_CB callback are not supported anymore.
- New: text formatting using FORMATTING and ADDFORMATTAG attributes on Windows and GTK. New attribute OVERWRITE when using text formatting.
- New: ALL and NONE values for SELECTION attribute.
- New: SCROLLTO attribute. New attributes SCROLLTOPOS, CARETPOS and SELECTIONPOS using 0 based character position. New function **IupTextConvertXYToChar** to convert (x,y) coordinates in (lin, col, pos) character positioning.
- New: SPIN, SPINVALUE, SPINMIN, SPINMAX, SPININC, SPINALIGN and SPINWRAP attributes. New SPIN_CB callback. The **IupSpin** control is now obsolete.
- New: VISIBLECOLUMNS, VISIBLELINES attributes gives much better control over size than the SIZE attribute.
- Changed: **IupMultiline** is now implemented as **IupText** with MULTILINE=YES.
- Changed: **ATTENTION - VERY IMPORTANT -** the ACTION callback in **IupText** now does NOT process extended keys anymore. It is called only if the text is edited, and key=0 if it is not a valid character. The callback now is called before the text is updated on screen.
- Changed: the SELECTION and CARET attribute on Windows do NOT change the focus anymore. The NC attribute now only restricts keyboard input.
- Changed: added support for BUTTON_CB and MOTION_CB callbacks. BUTON_CB can return IUP_IGNORE so the default processing will be ignored.
- Changed: CARET_CB now includes 0 based character position.
- Changed: **ATTENTION -** the Natural Size does not uses the text contents anymore. To control the Natural Size use the SIZE/RASTERSIZE attributes, or VISIBLECOLUMNS/VISIBLELINES attributes, or EXPAND.

List

- New: APPENDVALUE, CANFOCUS, COUNT, DRAGDROP, INSERTITEMn, REMOVEITEM, TOPITEM, SPACING, VISIBLECOLUMNS, VISIBLELINES attributes.
- New: BUTTON_CB, DBLCLICK_CB, DROPDOWN_CB, DROPFILES_CB, MOTION_CB callbacks.
- New: **IupListConvertXYToItem** function.

Other Standard Controls

- New: INVERTED and TICKSPOS attributes for **IupVal**.
- New: PADDING, VALUE_HANDLE, VALUEPOS, MULTILINE and TABIMAGE attributes for **IupTabs**.
- New: control **IupProgressBar**, similar to **IupGauge** but with the text.
- Changed: **IupFrames** now are native parents of their children.
- Changed: **IupVal** implemented as a native control. Attributes HANDLER_IMAGE and HANDLER_IMAGE_INACTIVE are not supported anymore.
- Changed: **IupCbox** is not based on **IupCanvas** anymore.
- Changed: **IupTabs** implemented as a native control. Attributes ALIGNMENT, FONT_ACTIVE, FONT_INACTIVE, TABSIZE and REPAINT are not supported anymore.

Additional Controls

- New: focus feedback and keyboard control for **IupColorbar**.
- Changed: **IupControlsClose** is now deprecated. Declaration still remains for compatibility, actual function does nothing.
- Changed: the NO_COLOR attribute is deprecated, now it simply sets the BGCOLOR attribute in **IupCells**.
- Changed: in **IupColorBrowser** moved from HLS to HSI, added support for resize, anti-aliasing, support for BGCOLOR attribute, feedback for ACTIVE attribute, and feedback for focus. New HSI attribute. New support for mouse wheel to change Hue. New support for PgDn and PgUp keys to change Hue.
- Changed: **IupTabs** and **IupVal** are NOT part of the additional controls anymore. They are now standard controls using native elements.
- Changed: renamed MARGIN attribute to PADDING in **IupGauge**. **IupGauge** is deprecated in favor of **IupProgressBar**.
- Changed: An **IupGLCanvas** when inside an **IupFrame** in Win32 will now work normally. But the dialog COMPOSITE attribute must be NO for hardware acceleration on Windows.

Menus

- New: HIDEMARK, AUTOTOGGLE and TITLEIMAGE attributes for **IupItem**.
- New: BGCOLOR support for **IupMenu**.
- New: Submenu now supports the IMAGE attribute.
- New: RADIO attribute for **IupMenu**.
- Changed: on GTK to have a menu item that can be marked you must set the VALUE attribute to ON or OFF, or set HIDEMARK=NO, before mapping the control.
- Changed: The HIGHLIGHT_CB, OPEN_CB and MENUCLOSE_CB callbacks now work normally for popup menus. HIGHLIGHT_CB is called for items and submenus.
- Changed: OPEN_CB and MENUCLOSE_CB are defined for menus, but it is checked at the parent submenu for backward compatibility with IUP 2.x.
- Changed: TITLE for submenus can now be changed after the element is mapped.
- Changed: Children can be added or removed from menus even after the menu is mapped.
- Changed: menus can now be dynamically changed even after mapped.

Images

- New: support for 24 and 32 bpp images using **IupImageRGB** and **IupImageRGBA** constructors.
- New: "UPARROW" cursor on Motif. New cursors "RESIZE_NS" and "RESIZE_WE". Updated cursor documentation with pictures of all pre-defined cursors.
- Changed: the automatic generation of inactive images for a more smooth one, still using a modified version of the background color to create the disabled effect.
- Changed: **IupImageLibOpen** will now only register names, but will not load the images. New 32bpp images for Windows. GTK aliases are also available. Many new images. **IupImageLibClose** removed, loaded images will now be automatically unloaded.

Keyboard

- New: MODKEYSTATE global attribute in all drivers.
- New: key definitions: K_acute, K_ccedilla, K_Print, K_Menu.
- New: key definitions for the system key modifier K_y*. on Windows this is the Windows key and in Mac this is the Apple key.

- New: CANFOCUS attribute for **IupButton**, **IupToggle**, **IupText**, **IupCanvas** and **IupVal**.
- Changed: SHIFTKEY and CONTROLKEY are now available in all drivers.
- Changed: Removed the conflicts: K_BS=K_cH, K_TAB=K_cI and K_CR=K_cM. New key code macros **iup_isShiftXkey**, **iup_isCtrlXkey**, **iup_isAltXkey** and **iup_isSysXkey**.

History of Version 2.x

Migration Guide
IUP 2.x to IUP 3.x

Critical Changes (from 2.x to 2.7/3.0)

All critical changes were packed in version 2.7 so you can prepare your code to work with both 2.7 and 3.0 versions. And you will be able to alternate between both versions without having to add "ifdef"s to your code. The differences in the "iup.h" header file from 2.7 to 3.0 should contains only the new features introduced in 3.0.

IupOpen function declaration now include command line arguments used by X-Windows and GTK - The most important change is the signature of the **IupOpen** function. It was changed to include the main function arguments. The GTK and Motif toolkits use them. In IUP prior to version 2.7 they were ignored for Motif. In Windows they are always ignored. If for some reason you do not have access to the main function arguments you can use NULL in **IupOpen**. As a general rule the change is:

```
IupOpen()  >>  IupOpen(&argc,&argv)
```

You will also have to search&replace a few things in your source code:

```
the attribute "WIN_SAVEBITS"  >>  "SAVEUNDER"
the function IupGetType  >>  IupGetClassName
```

Although the following were considered obsolete in IUP 2.6, their backward compatibility code were removed in 2.7. So you may have to search&replace for:

```
the attributes "MOTIF_FONT" and "WINFONT"  >>  "FONT"
the value of the attribute "CURSOR" =  "IUP"  >>  "HELP"
the definition IUP_ANYWHERE  >>  IUP_CURRENT
the constructor IupColor (removed)  >>  use the color value
```

The "cdiup" and "cdluaiup" libraries moved from CD to IUP under the name "iupcd" and "iupluacd" - Also you will have to change your makefile or IDE project because we changed some library names to solve the cross dependencies between IUP, CD and IM libraries.

Strategic Changes (from 2.7 to 3.x)

All the changes described here are backward compatible with 2.7. So after doing them you will still be able to go back to 2.7.

Some global attributes like DEFAULTFONT, *BGCOLOR and *FGCOOR are now obtained from the system instead of hardcoded, this affects mainly applications in Windows were the hardcoded DEFAULTFONT was "Tahoma, 8" and the user changed the default font or used the Large Fonts option. If your dialog is too big in the new font then you can simply set DEFAULTFONTSIZE to force a smaller value.

The following headers were deprecated iupcb.h, iupcells.h, iupcolorbar.h, iupdial.h, iupgauge.h, iupmatrix.h, iuptree.h - they now simply include iupcontrols.h. You can replace them by iupcontrols.h in your code.

The following headers were deprecated iupcbox.h, iupsbox.h, iupgetparam.h, iupspin.h, iuptabs.h, iupval.h - they now simply include iup.h. You can remove them from your code.

The ACTION callback in **IupText**/**IupMultiline** now does NOT process extended keys anymore - the callback is called only if the text is edited, and key=0 if it is not a valid character. In 2.x the key parameter were used for some navigation keys, but now is used only for keys associated with characters. This is the most impacting change from 2.7 to 3.0, because some functionality in your application could stop working. Use the K_ANY or K_* callbacks instead to process navigation keys.

The Natural Size of **IupText**/**IupMultiline** does not uses the text contents anymore - to control the Natural Size use the SIZE/RASTERSIZE attributes, or the VISIBLECOLUMNS/VISIBLELINES attributes, or the EXPAND attribute. This will avoid the automatic resize of the **IupText**/**IupMultiline** if its content is changed by the user and the size of the dialog is changed so the layout is recalculated.

Now in **IupCanvas** the scrollbar parameters X/YMIN, X/YMAX and X/YLINE are updated only when DX/Y are updated. POSX/Y will only update the position of the scrollbar. In version 2.x was necessary to set POSX/Y to update those parameters.

The BGCOLOR attribute is now ignored in **IupLabel**, **IupFrame**, **IupToggle** (for the text background) and **IupVal**. They will use the background color of the native parent. **IupFrame** can has a color background when not using TITLE, and BGCOLOR is set before map.

The **IupItem** in GTK must have its VALUE attribute defined (ON or OFF) **before** mapping - so it can have the check mark, or define HIDEMARK=NO. If not done the item will not be checkable.

The new **IupTabs** does not supports the inactive tab feedback. So the tabs will be always active, although its children will be sucessfully disabled. The return value of the TABCHANGE_CB callback is not processed anymore. The most impacting feature is the TABORIENTATION attribute that has limited support in the native controls.

In **IupMatrix** the selection is made only using the mouse, pressing a key will NOT clear the selection anymore. You can still do that setting MARKED=NULL in the K_ANY callback.

In **IupTree** DEPTH is now a read-only attribute, use the INSERT* attributes to properly add nodes. The SELECTION_CB and MULTISELECTION_CB callbacks now ignore their return value. Now you can only add nodes to the tree after it has been mapped to hte native system. NAMEid attribute renamed to TITLE, old attribute still works but will be removed in future versions.

History of Changes in Version 2.x

CVS (17/Jun/2009)

General

- Changed: the **IupOleControl** in Lua will not automatically initialize LuaCOM anymore. The application must manually call "elem:CreateLuaCOM()". The previous initialization was incorrect (thanks to Ross Berteig).
- Fixed: parameter checking and the return value in Lua for **IupListDialog** when type=2.
- Fixed: the return value for **IupGetText** when the user canceled.
- Fixed: The 32 bits version of the IupLua console in Windows XP64 was not working.
- Fixed: IupLua initialization when retrieving the argc/argv arguments for **IupOpen**. (Thanks to Ross Berteig)
- Fixed: action callback return value in Lua for the **IupGetParam** dialog. (Thanks to Zhiwei)

Version 2.7.1 (15/Dec/2008)

General

- Fixed: the iuplua51 makefile where not using the g++ linker, so require"imlua" failed.
- Changed: removed csh dependency from make_uname scripts.
- Changed: removed IupSpeech from source and documentation.

Motif

- Fixed: IupOpen was crashing if used with NULL parameters.
- Fixed: DIRECTORY attribute in IupFileDlg when set to NULL did an invalid memory access.

- Fixed: invalid default value for SCROLLBAR attribute in IupMultiline and IupList.
- Fixed: size of tips window when displaying a multiline string.

IupControls

- Fixed: some frame lines where not drawn in IupMatrix.

Version 2.7 (14/Oct/2008)

General

- Changed: **INCOMPATIBILITY** - **IupOpen** function declaration now include command line arguments used by X-Windows and GTK.
- Changed: **INCOMPATIBILITY** - "IUP" cursor in Windows renamed to "HELP" cursor.
- Changed: **INCOMPATIBILITY** - WIN_SAVEBITS renamed to SAVEUNDER.
- Changed: **INCOMPATIBILITY** - removed old "MOTIF_FONT" and "WINFONT" attributes. Use only the "FONT" attribute.
- Changed: **INCOMPATIBILITY** - removed old IUP_ANYWHERE and IupColor definitions.
- Changed: **INCOMPATIBILITY** - **IupGetType** renamed to **IupGetClassName**.
- Changed: **IMPORTANT** - all functions that receive a constant string now has the "const" modifier for the string parameter declaration.
- Changed: **IMPORTANT** - Copyright notice modified to reflect the registration at INPI (National Institute of Intellectual Property in Brazil). License continues under the same terms.
- Changed: **IMPORTANT** - the support services (Downloads, Mailing List and CVS) moved from LuaForge to SourceForge.
- Changed: All dll8 and dll9 DLLs now have a Manifest file that specifies the correct MSVCR*.DLL.
- Changed: Makefiles for UNIX now uses a compact version of Tecmake that does not need any installation, just type "make".
- Changed: removed "INCLUDE" parameter for FILE_CB callback in IupFileDlg.
- Changed: improved automatic inactive image generation.
- Changed: premake files are used now only internally and were removed from the distribution.
- Changed: IupLua3 libraries are not included in the distribution anymore. They are only available in source code or internally at Tecgraf.
- Changed: All Lua samples now have the extension .wlua, and contains require"iuplua" and iup.MainLoop() in the code. Thanks to Ryan Pusztai.
- Changed: added traceback information to error message dialog in IupLua. Thanks to Fred Abraham.
- Changed: The IupLua Console now must include require commands for any additional library.
- Fixed: IupView image export in C format.
- Fixed: removed MARGIN from IupFrame documentation. IupFrame does not have a MARGIN attribute.
- Fixed: removed MARGIN from IupZbox documentation. IupZbox does not have a MARGIN attribute.
- Fixed: SYSTEM global attribute in Windows, when running Windows Vista.
- Fixed: Improved visual appearance and ticks of bar mode in IupPPlot.
- Fixed: missing IupMessagef export in the DLL.
- Fixed: LEDC generated code for 64-bits.
- New: **IMPORTANT** - the "cdiup" and "cdluaiup" libraries moved from CD to IUP under the name "iupcd" and "iupluacd". But headers and documentation remains on the CD package. Function names were NOT changed. This change eliminates a cross-dependency of IUP and CD, now only IUP depends on CD.
- New: "iupluaimglib" library so require"iupluaimglib" can be used to dinamically load the image library.

Windows

- Fixed: invalid memory access when set FONT to NULL.
- Fixed: CARET position when a selection is interactively changed or when the caret is at the begining of the selection in IupText, IupMultiline and IupList.
- Fixed: TABSIZE IupMultiline attribute scale convertion.
- Fixed: invalid character inserted in IupMultiline when opening a dialog from a Ctrl+key combination.

Motif

- Fixed: Removed X run time warning when creating a list.

IupControls

- New: FRAMEVERTCOLORL:C, FRAMEHORIZCOLORL:C and CHECKFRAMECOLOR attributes for IupMatrix.
- Fixed: EXTENDED_CB callback was never called in IupColorbar.
- Fixed: invalid memory access in IupTree when using images with a color index greater than 128.]
- Fixed: invalid memory access in IupTabs when all tabs are disabled and a next or previous tab button is pushed.
- Fixed: invalid memory access in IupColorbar.
- Fixed: invalid call to CLICK_CB when resizing column in IupMatrix.

Version 2.6 (26/Nov/2007)

General

- Changed: SELECTION attribute in IupText now accept values in reverse order.
- Changed: IupView improvements. New functions: "Save All Images"; "Save All Images in One File". Changes: "Import Image" can load multiple images in Windows; "Save Image" allow to save in GIF format.
- New: SCROLLBAR attribute for IupMultiline and IupList.
- New: WORDWRAP attribute for IupMultiline.

Windows

- New: "INCLUDE" parameter for FILE_CB callback in IupFileDlg.
- Fixed: FONT creation when system uses a non ANSI charset.

Motif

- Fixed: FONT attribute internal storage.
- Fixed: IupMapFont interpretation of the size value to use points in X-Windows Logical Font Description format (XLFD).

IupControls

- New: new parameter for IupGetParam to specifiy a file name string that can be changed using a file selection dialog. Thanks to Flavia Anjos. New interval step for real and integer interval.
- Fixed: for all additional controls the used font follows strict the FONT attribute. Previously for some of the controls the CD default font were used causing an inconsistency with the control size calculation.
- Fixed: ACTIVE update in IupVal.
- Changed: in IupPPlot ACTIVE attribute renamed to CURRENT to avoid conflict with the IupCanvas ACTIVE attribute. Fixed DS_MODE and DS_EDIT return values. Fixed DS_EDIT when set to "NO" from a previously set to "YES".
- Changed: moved IupSbox, IupCbox and IupSpin to the core library. They do not depend on the CD library.

IupMatrix

- Changed: BGCOLORL:C, FGCOLORL:C and FONTL:C are now handled different for title columns and title lines. When you set the color or font of a full line/column it will not affect the title line/column except when that line/column is the title line/column (lin=0 or col=0). Individual cell colors are still handled independently.
- New: RASTERWIDTHn and RASTERHEIGHTn attributes.
- Fixed: EDITION_CB called with invalid self parameter.
- Fixed: DROPSELECT_CB called after dropdown list is hidden.

IupLua

- Fixed: missing IupCells and IupColorbar initialization in iupcontrolslua_open.
- New: added LuaGL binding to the IupLua console executable. So OpenGL commands can be used in Lua.

Version 2.6 RC2 (10/May/2007)

General

- New: function IupUpdate to force a redraw of the element and its children.
- New: function IupExitLoop to exit the current message loop. It is equivalent of returning IUP_CLOSE in a callback.
- Changed: now for the IupList when DROPDOWN=Yes the size of the dropped list will expand to include the largest text.

Version 2.6 RC1 (15/Apr/2007)

General

- New: functions IupGetChild, IupGetAllAttributes.
- New: CLIPBOARD attribute with COPY, PASTE and CUT values for IupText and IupMultiline.
- New: control IupPPlot that uses the PPlot library to draw 2D plots. Thanks to Marian Trifon.
- Changed: LEDC now supports IupCells, IupCbox, IupOleControl and IupSpin.
- Changed: IupMultiline and IupText size calculation. When EXPAND is different than NO or NULL, the control will ignore its contents when calculating the control size if SIZE or RASTERSIZE is not set. So now if text is larger than the multiline and EXPAND is set, the multiline will not expand to include its contents when the dialog is expanded. In this case the multiline will be expanded only what the dialog allows it to expand.
- Changed: size update when FONT is set. Now to update the control size IupRefresh must be called.
- Fixed: Added missing documentation of IupGetParent.
- Fixed: caps lock key codes.
- Fixed: Added missing IupSetAttributeHandle and IupGetAttributeHandle exports in the DLL.

Windows

- Changed: Resource files moved from "iup/lib" to "iup/etc".
- Changed: IupFileDlg attributes FILE and DIRECTORY in Windows to accept paths containing also "/".
- Fixed: dialog activation after IupPopup.
- Fixed: IUP_CLOSE return in K_ANY and K_* callbacks.
- Fixed: WHEEL_CB parameters x and y.
- Fixed: IupPopup for menus when used in the Tray if there is no visible dialogs.
- Fixed: FONT attribute initialization when control is not mapped yet. Affected mainly controls inside other controls.
- Fixed: FONT attribute parse when value is invalid.

Motif

- New: TOPLEVEL global attribute.
- Changed: default IupHelp application in Linux to "firefox".
- Changed: some attributes were updating the size of the control in the dialog. Now to update the control size IupRefresh must be called.
- Fixed: Idle processing.
- Fixed: return value of IupLoopStep.
- Fixed: invalid resize of IupList when COMBOBOX=YES and an element is added dynamically.

IupLua

- New: convertion to string for an Ihandle. Now returns "IUP(*type*): *address*", for example "IUP(dialog): 08C55240".
- Changed: IupLua5 executable in Windows to enable GDI+ in CD library.
- Changed: IupLua3 libraries names changed to include "3" as a suffix.
- Fixed: Added missing IupGLIsCurrent binding.
- Fixed: error message management when inside a callback in Lua 5.
- Fixed: error handling in iuplua_dofile and iuplua_dostring.
- Fixed: the second ihandle parameter inside the callabacks in Lua 3: DROP_CB, DROPSELECT_CB and TABCHANGE_CB.
- Fixed: conflict in dialog resize attribute with resize callback from canvas in Lua 3.
- Fixed: getattribute metamethod when value is not a number or string before calling GetHandle to check if it is a handle.
- Fixed: setattribute metamethod when value is stored in C now is also set to nil in Lua to avoid old invalid values in Lua.
- Fixed: IupAlarm optional parameters in Lua 3.
- Fixed: missing edit_cb callback definition for IupList in Lua 5.
- Fixed: Lua object memory management when destroy is called.

IupMatrix

- New: RELEASE_CB mouse callback.
- Changed: DRAW_CB callback to add the CD canvas as the last parameter. Now the canvas is also available for CDLua.
- Fixed: BGCOLOR and FGCOLOR for full lines or full columns in titles (L:* or *:C).
- Fixed: BGCOLOR for titles and empty area to use the parent's BGCOLOR instead of the dialog BGCOLOR.
- Fixed: BGCOLOR_CB and FGCOLOR_CB in Lua when IUP_IGNORE is returned.
- Fixed: setting VALUE attribute when the cell is being edited.
- Fixed: redraw when resizing collum and the scroolbar is added to the canvas in Windows.
- Fixed: redraw in SunOS after editing the cell.

Other IupControls

- Changed: in IupTabs, when next or previous tab is selected using the arrow buttons or arrow keys, inactive tabs are skipped.
- Changed: CD calls to use the new CD API available only in CD version 5.0. So IUP will not be compatible with old CD versions.
- Changed: Because of the new parameter of DRAW_CB callback in IupMatrix, the IupControls Lua binding now depends on the CD Lua binding.
- Fixed: F2 key processing to rename a node in IupTree.
- Fixed: focus change when changing the active tab in IupTabs.
- Fixed: BUTTON_PRESS_CB and BUTTON_RELEASE_CB binding in Lua 3 for IupDial and IupVal.
- Fixed: IupTree rename box position when using scrollbars.

IupGLCanvas

- New: SHAREDCONTEXT attribute.
- Fixed: Added missing DLL export IupGLIsCurrent.

Version 2.5 (31/Mar/2006)

General

- IMPORTANT: New functions IupSetCallback and IupGetCallback to register callbacks without using a global name. IupGetFunction and IupSetFunction are still working, but are not used internally anymore. The new functions speed up the performance of callbacks, and reduce to zero name conflicts for callbacks in the global name table. It is recommended that the applications should replace IupSetFunction and IupGetFunction by IupSetCallback and IupGetCallback. IupLua applications are automatically benefited.
- IMPORTANT: Applications that overload internal callbacks of the additional controls (like IupMatrix and IupTree) must now use IupSetCallback and IupGetCallback to do the overloading. And as before these callback can not be overloaded in Lua.
- IMPORTANT: removed the support for callback inheritance. Now callbacks can only be set in the own element. The only execption is the K_ANY and the K_* callbacks that continues to be

propagated to the parent of the element with the keyboard focus. (This was a not very usefull feature, with very few uses. But slows a lot calback management in C and in Lua. With the new IupGetCallback we were able to remove the inheritance mecanism for callbacks.)
- Changed some function declarations of the main API, some now use "const char*" in their declaration.
- Changed global attributes now are stored only if not processed by the driver.
- IMPORTANT: Changed the definition of Icallback to a simple one without the variable arguments. Fixed canvas callback parameters, in the documentation is float, but with the old Icallback definition the compiler used double. Now must be float.
- Changed all the internal attributes now start with the preffix "_IUP".
- Changed the default limit for text in IupText and IupMultiline to be 2^{31}.
- New canvas callback FOCUS_CB.
- New helper function IupSetAttributeHandle to associate Ihandle* to attributes using automatic names. Instead of using IupSetHandle and IupSetAttribute with a new creative name, this function automatically creates a non conflict name and associates the name with the attribute. Also new function IupGetAttributeHandle.
- New "Pause" button in the IUP Image Library.
- Fixed the MULTISELECT_CB callback of the IupList so it does not need that the ACTION callback is also defined.
- Reviewed the popup dialog management. So we improve the behavior of the IupShow of other dialogs after a IupPopup, and a new possibility to safely cascade popups.

Windows

- IMPORTANT: Global attribute WIN_DEFAULTFONT renamed to DEFAULTFONT.
- Fixed attribute PLACEMENT=NORMAL when the dialog in minimized or maximized.
- Fixed IupPopup for menus, when the menu item callback returned IUP_CLOSE, the return value is now processed and the application is closed.
- Change WOM_CB and be set also for the dialog.
- Changed CoInitialize to CoInitializeEx[COINIT_APARTMENTTHREADED] and InitCommonControls to InitCommonControlsEx[ICC_WIN95_CLASSES] in IupOpen.

IupControls

- Changed the GETFOCUS_CB and KILLFOCUS_CB callbacks for the additional controls IupMatrix, IupVal and IupDial, now can be set without affecting their implementation.
- Changed the K_ANY for the additional controls IupTree, IupSpin and IupColorBrowser, now can be set without affecting their implementation.
- New DOUBLEBUFFER attribute for IupTabs. Default is YES. If NO will disable the double buffer. This may solve a slow Tabs redraw in UNIX when the a Tab contains many controls.
- New IupVal attributes HANDLER_IMAGE and HANDLER_IMAGE_INACTIVE that allow the use of images to replace the handler. Thanks to Rodrigo Espinha.
- Reviewed and optimized iupMask code. Added new callback MATCH_CB.

IupMatrix

- IMPORTANT: Callbacks ACTION and SCROLL_CB were renamed to ACTION_CB and SCROLLTOP_CB to avoid conflict with the IupCanvas callbacks also inherited by the IupMatrix.
- IMPORTANT: You can not automatically override the KEYPRESS_CB callback anymore. You must save the original callback and call it from inside your own.
- IMPORTANT: Now when in callback mode much less memory will be allocated. Also the new callbacks MARK_CB and MARKEDIT_CB can be used to control the selected cells in callback mode.
- Fixed some string buffer sizes to handle very large matrices.
- Fixed IupGetAttribute for the VALUE attribute when using callback mode and retrieving colum or line title values ("0:C" or "L:0").
- Changed "matrx_img_cur_excel" to "IupMatrixCrossCursor". Old name is still available.

IupTree

- IMPORTANT: The IupTree implementation now uses the KEYPRESS_CB callback. The K_ANY override support was removed. The K_ANY callback can be used normally. If the application was using the KEYPRESS_CB, now it must override it manually, you must save the original callback and call it from inside your own.
- Change the appearance in Windows and Motif are now the same. Both systems look like the previous Windows implementation with a white background and some small enhancements.

IupLua

- IMPORTANT: IupLua3 now supports IupLua5 names. Old IupLua3 names still work, but now all the samples for IupLua5 also work in IupLua3. The documentation and the examples for the old names were removed from the manual pages. Old applications using IupLua3 can use the old names or the new names. This will make easier to old applications migrate their code to Lua 5. All Lua examples were re-tested and fixed.
- IMPORTANT: In IupLua3 the callbacks in C are registered only when the application register the callback in Lua, just like in IupLua5.
- IMPORTANT: IupColorBrowser name changed in IupLua3 from "iupcb" to "iupcolorbrowser".
- Fixed documentation of IupGetAllDialogs and IupGetAllNames. Fixed implementation to match the documentation.
- Fixed IupTimer old callback name in IupLua3.
- Fixed DROPFILES_CB canvas callback can be now used in Lua for the controls based in IupCanvas, like IupMatrix and IupTree.
- Fixed parameters of the canvas action and scroll_cb callbacks in Lua 5.
- Fixed missing FILE_CB callback in Lua.
- Changed all the additional controls now can have the K_ANY, GETFOCUS_CB and KILLFOCUS_CB callbacks without affecting their internal implementation.
- Changed Lua 5.1 "require" can now be used for all the IupLua 5.1 libraries, but the full library name must be used. For example: require"iuplua51", require"iupluacontrols51".
- Documented the IupLua 5 architecture.
- Reviewed and reorganized IupLua3 and IupLua5 code, also cleaned and simplified. In IupLua3 callbacks are now set only if they are set by the application.
- Changed IupClose can now be called from Lua in Lua 5.
- Reviewed and improved the interchange of Ihandle between C and Lua. The documentation was updated with all the possibilities.

Version 2.4 (12/Dec/2005)

General

- New attribute ZORDER to change the zorder of any control or dialog.
- New 3STATE attribute for IupToggle to enable a three state text toggle.
- Reviewed and improved the creation of controls, so they can be added to an already created dialog.
- Reviewed and improved the natural size estimation for each standard controls. The estimation now is the same for Windows and Motif with some minor differences for border and scrollbar sizes. All the controls can have sizes bigger or smaller than the natural size using SIZE or RASTERSIZE attributes (natural size is the size of the control that fits all of its contents).
- Improved FULLSCREEN IupDialog attribute in Windows and Motif, so the application can set fullscreen and then restore to normal state any time.
- New attribute FLAT for IupButton to create a button with mouse over activation (Windows and Motif).
- New MULTISELECT_CB callback for IupList. It can replace the action callback for multiple selection lists.
- Fixed names of headers, initialization functions and libraries that did not have the "iup" prefix. Headers "iupolecontrol.h", "luacontrols.h" and "luagl.h" changed to "iupole.h", "iupluacontrols.h" and "iupluagl.h". Private headers and declarations removed from "iup/include" folder. Functions controlslua_open, gllua_open and iupluaim_open changed to iupcontrolslua_open, iupgllua_open and iupimlua_open.
- New documentation of the IupOleControl control, including a sample and Lua bindings. Thanks to Vinicius Almendra.
- New function IupRefresh to update the size and layout of controls after changing size attributes.
- Exported the internal functions: IupZboxv, IupHboxv, IupVboxv and IupMenuv.
- Fixed several memory leaks. Thanks to Visual Leak Detector.
- IupView application can now save imagens in C source code format.
- New additional library with several pre-defined images for buttons and labels. See IupImageLib.
- Optimization flags now are ON when building the library in all platforms.
- Now all the predefined dialogs consult the global attribute IUP_ICON.
- Missing key definitions: K_sDEL and K_sINS. This prevented the Del key to work when CAPSLOCK was active in some controls.
- Changed IUP_QUIET environment variable now default is YES.

Windows

- Support for MDI (Multiple Document Interface). See IupDialog documentation.
- Fixed IupLabel with IMAGE with invalid focus.
- New SUNKEN attribute for IupFrame.
- Fixed appearance of IupLabel with IMAGE when ACTIVE=NO.
- Fixed initial value in the IupList when EDITBOX=YES.
- Now it is not necessary anymore to use the "iup.rc" file for the HAND cursor. It is now build in.
- New value for PLACEMENT attribute, FULL to position the client area of the dialog in fullscreen.

- IupButton and IupToggle with images using Windows XP Visual Styles now uses a styled border. See IupButton documentation for samples.
- Missing documentation of ENTERWINDOW_CB and LEAVEWINDOW_CB for IupButton.
- Fixed button draw with BGCOLOR and empty text.
- New COMPOSITED attribute to create a window with an automatic double buffer for all controls.
- New LAYERED and LAYERALPHA attributes to set and configure layered windows using transparency.
- Fixed image offset in IupButton.
- Fixed invalid redraw for IupLabel using an IupImage when inside a IupTabs or IupSbox.
- Added an "ifndef IUP_NO_ABNT" enclosing the ABNT keyboard management so it will be easier to ignore this code from the makefile.
- Default FONT in Windows XP is now the Tahoma font.
- BGCOLOR for canvas was not being updated correctly when changed after canvas creation.

Motif

- SHOWDROPDOWN now works also in Motif.
- Removed horizontal scrollbar parameter from simple IupList (DROPDOWN=NO and EDITBOX=NO) to made it compatible with the other lists (including the simple IupList in Windows).
- Fixed KILLFOCUS_CB and GETFOCUS_CB for IupList with DROPDOWN=YES or EDITBOX=YES.
- Fixed invalid IupList resize when DROPDOW=Yes after inserting elements in the list.
- New BACKINGSTORE IupCanvas attribute so the backing store can be disabled.
- Changed IupToggle with IMAGE and IMPRESS to behave like in Windows, where the button border is always shown.
- Fixed error in menu item initialization.

IupControls

- IMPORTANT: for best results CD version 4.4 should be used.
- Fixed IupSpin keyboard response and mouse press & hold response.
- New MULTISELECTION_CB callback for IupTree.
- New IupCells control. It is an application controlled matrix. More simple and faster than IupMatrix. Can also span cells. Thanks to Andr Clinio.
- New IupCbox control for concrete layout positioning.
- Fixed IupTabs tab activation using mouse. It could activate a different tab using button press in one tab and button release in another tab.
- Fixed spin buttons were not calling the user callback in IupGetParam.
- Fixed IupVal non effective increment using keyboard when at minimum value.
- Fixed invalid IupSetAttribute for scrollbar parameters in IupTree that affects navigation of two or more trees in the same application.
- Fixed keyboard usage when CAPSLOCK is active for IupVal, IupTabs and IupDial.
- New functions iupMaskRemove and iupmaskMatRemove to remove the iupMask from a control.
- New RENAME action attribute for the IupTree.
- New attribute TABORIENTATION to change the tab text orientation. The active tab text is now bold.
- Changed CARET and SELECTION attributes of the IupTree when using an in-place rename text box, to RENAMECARET and RENAMESELECTION. This will avoid conflict with the SELECTION_CB callback in IupLua3.

IupMatrix

- Redefined REDRAW policy to a more precise and effective one. No redraw is done when the application sets cell, line or column graphics attributes attributes: **0:0**, **0:C**, **L:0**, **L:C**, **ALIGNMENTn**, BGCOLORL:*, BGCOLOR*:C, BGCOLORL:C, FGCOLORL:*, FGCOLOR*:C, **FGCOLORL:C**, F**ONT**L:*, F**ONT***:C, **FONTL:C**. Global and size attributes always automatically redraw the matrix.
- Improved double click editing in Motif. Since OpenMotif 2.2.3 the double click to edit the cell works fine. For previous version there is still a workaround to show the controls and the need to click again in the control so it get the focus.
- All the edition mode code were rewritten and reorganized in a separated module. Any old code was removed and cleaned.
- Small change in focus feedback, its area was reduced to two pixels in each cell border.
- Cell focus management code reorganized to a more simple and efficient version.
- New SORTSIGN*C* attribute to show a sort sign (up or down arrow) in the column C title.
- New drawing in double buffer mode to minimize flicker.
- Fixed dropdown feedback drawing.
- Fixed focus feedback after double click editing.
- The alignment of the text in a cell with a dropdown feedback now considers the horizontal space occupied by the feedback.
- The DRAW_CB callback drawing area now does not includes the focus feedback area if HIDEFOCUS=NO (the default).
- NUMCOL_VISIBLE and NUMLIN_VISIBLE now when retrieved returns the current number of visible lines.
- Fixed problem after trying to edit a non editable cell the focus gets lost.
- Reviewed documentation and behavior of marks.

IupLua

- IupLua5 source code is now 100% compatible with Lua 5.1.
- The iuplua binding and all its libraries can now be dinamically loaded in Lua 5. IupOpen will be automatically called.
- iupkey_open can now be called from Lua 5, using iup.key_open.
- New IupGetParam binding.
- Changed the keys definitions (K_*) in Lua so now they are exactly the same as the definitions in C.
- Fixed invalid IupGetAllNames in IupLua5. Fixed missing IupGetAllNames binding in IupLua5.
- Fixed IupTree EXECUTELEAF_CB callback in IupLua5. It was expecting an invalid extra parameter.
- Fixed error in IupTabs memory initialization in IupLua5.
- Fixed missing IupGetText binding.
- Fixed missing pre-defined masks for iupMask.
- Fixed missing isxkey macro binding.
- Fixed missing callback scroll_cb in IupLua3.
- Fixed missing IupVersion documentation and binding.
- Fixed IupSetGlobal and IupStoreGlobal in IupLua5.

Version 2.3.1 (18/Apr/2005)

General

- New support for 64-bits Linux.
- New global attribute DLGBGCOLOR.
- Changed the KEYPRESS_CB and K_ANY callback are now compatible with Portuguese Brazilian ABNT keyboard layout in Windows and Linux.
- Changed key names **K_quoteright** and **K_quoteleft** renamed to **K_apostrophe** and **K_grave**, but there are backward compatible defines.
- Fixed IupOpen/IupClose for correct initialization/de-initialization.
- Fixed IupGetGlobal to retreive first from the driver.
- Fixed IupDestroy for correct memory deallocation.
- Fixed IupLoadImage to include BGCOLOR information. New function IupSaveImage.
- New Guide / C++ Usage section in the documentation, with additional C++ wrappers contributed by some users. Thanks to Danny Reinholds, Sergio Maffra and Frederico Abraham.

Windows

- Fixed K_ANY duplicate calls for some keys.
- Fixed popup menu bug. Sometimes when selecting an item the callback was not called.
- Changed IupText and IupMultiline now can have the ALIGNMENT attribute.

Motif

- Fixed use of variable parameter arguments in Motif calls to correct 64-bits compatibility.
- Fixed some small bugs in IupDestroy. GETFOCUS_CB callbacks were called during dialog destroy. Menu bars were incorrectly destroyed.

IupControls

- Changed IupGetParam now uses only the number of lines to determine the number of parameters. The last 0 is not necessary anymore.
- Fixed bug in IupColorBrowser destroy.
- Fixed IupTree initialization for LED usage.
- New IupTree feature to rename a node in place.
- New IupColorbar control. It is a palette of colors to allow the selection of primary and secondary colors. Thanks to Andr Clinio.

IupGLCanvas

- New function IupGLIsCurrent.

IupLua

- Fixed callbacks for IupDial in IupLua5.

IupView

- Fixed data initialization in Motif.
- New menu items to save images in individual LED and Lua text files, and in Windows ICON files.
- New menu item to load an image using IM.

Version 2.3 (16/Mar/2005)

General

- Download, Discussion List, Submission of Bugs, Support Requests and Feature Requests, are now available thanks to LuaForge site.
- New organization of the documentation.
- New MacOS X libraries using OpenMotif and gcc.
- New CARET_CB callback for the IupText, IupMultiline and IupList controls. It is called every time the caret changes its position.

Windows

- IMPORTANT: Now the canvas background color is only redrawn if the ACTION callback is not defined. When defined the application must draw all the canvas contents. This will optimize the redraw of canvas based controls and application canvases. The TRANSPARENT value for the BGCOLOR is not supported anymore.
- New attribute IMMARGIN to control the spacing between the border and the image in IupButton.
- Optimized the IupButton and IupLabel drawing when IMAGE is specified.
- Fixed incorrect stop for the IupTimer. Improved start and stop control.
- Flicker now is significantly reduced. CLIPCHILDREN=YES is now default. IupFrame background drawing optimized.
- New dialog attribute "CONTROL" that enable the embedding of the dialog inside another window. Used by LuaCOM to create OLE (ActiveX) controls implemented in Lua.
- New IupText attribute "PASSWORD" to hide the typed character.
- IUP is now compatible with Windows XP Visual Styles. See the Win32 driver documentation.

Motif

- Fixed invalid return value when retreiving the FONT attribute.
- Added backward compatibility code for Motif 1.2. Must edit makefile to add the file "src/mot/ComboBox1.c".

IupControls

- Missing support for IupList with EDITBOX=YES in iupMask.
- BGCOLOR for images were ignored in the IupTree.
- Now some matrix cell attributes are not inherited from parent. Like "L:C", "ALIGNMENT*", "FGCOLOR*", "BGCOLOR*", "FONT*", "WIDTH*" and "HEIGHT*", for optimization reasons.
- IupTree now uses double buffer for optimal drawing.
  To avoid flicker during resize in Windows, do not use it inside a IupFrame, and use CLIPCHILDREN=YES.
- New utility functions: IupTreeSetAttribute, IupTreeStoreAttribute IupTreeGetFloat, IupTreeSetfAttribute, IupTreeGetAttribute, IupTreeGetInt.
- New IupMatrix callback DRAW_CB to allow a custom drawing of the cell contents.
- New IupTree DRAGDROP_CB callback.
- New IupSpin and IupSpinbox utility functions.

IupLua

- Fixed ihandle_gettable in iuplua.lua when iupGetTable is nil when object is created in C.
  This affected the object returned by iup.LoadImage.
- Fixed Zbox children names initialization.
- Missing DROPFILES_CB callback management.
- Missing FGCOLOR_CB and BGCOLOR_CB callback management for the IupMatrix. The returned values order was inverted.
- Missing MAP_CB callback management for IupCanvas in IupLua3.

Version 2.2.2 (07/Oct/2004)

General

- Fixed bug in IupGetFile FILTER initialization.
- Improved IMINACTIVE automatic generation algorithm.
- New zip package for download with iup images in LED format.
- New application IupView to load and display LED files.
- Fixed some attribute storage in iupMask and IupGetParam. Fixed bug when several masks are used in the same dialog.
- Replaced the internal Lua4 code for a smaller hash table module. Thanks to Danny Reinhold.
- Fixed IupGetParam invalid memory access.
- IupNextField and IupPreviousField now only changes the focus for the checked toggle inside a radio.
- IupGetAttributes now returns the pointer address if attribute is a known internal pointer data.
- Now pressing Enter over a button activates it, even if it is not the DEFAULTENTER button.
- Esc and Backspace keys now will be translated even if CapsLock is active.

Windows

- New ENTERWINDOW_CB and LEAVEWINDOW_CB for buttons.
- Fixed double click for button, toggle and list were not being considered as two clicks.
- removed FLAT style from toggles with IMPRESS image. Fixed size of toggle with image.
- New attribute SHOWDROPDOWN to open the dropdown list programmatically.
- Removed a black border around IupMultiline and IupText.
- Removed the TABSTOP for non marked Toggles inside a Radio.
- Fixed invalid memory access when menu item is activated and all dialog controls are disabled.
- Fixed IupFileDlg ignored the x,y parameters of IupPopup.

Motif

- Enter in IupMultiline activated the DEFAULTENTER button instead of adding a new line.
- Fixed invalid memory access when set FONT to NULL.
- Fixed ACTION callback called for IupList when list contents were cleared.

IupControls

- IupTree and IupTabs did not propagate to the parent the K_ANY callback for non used keys.

IupMatrix

- The TITLEs, BGCOLORs, FGCOLORs and FONTs attributes were incorrectly set after a DELLIN, ADDLIN, DELCOL or ADDCOL.
- In Windows when the user double click a dropdown list now will start opened.
- The user callback scroll_cb was incorrectly registered.
- New "HIDEFOCUS" attribute to hide the focus mark when drawing.
- Now in MARK_MODE=CELL and MULTIPLE=YES you can click on the title area to mark a full line or collumn at once.
- New BGCOLOR_CB and FGCOLOR_CB callbacks.
- Fixed when MARKMODE=LIN/COL/LINCOL if the first cell in the line/column is selected the click in the title area was ignored.

IupLua

- Removed "print" debug calls in internal code.
- IupGetAttribute/iup.GetAttribute now returns an user data if attribute is a known internal pointer data.
- New IupGetAttributeData/iup.GetAttributeData that returns the data always as an used data.
- Fixed incomplete initialization of image object returned by IupLoadImage.

Version 2.2.1 (25/Aug/2004)

General

- Fixed some minor bugs introduced in version 2.2.
- Fixed HTML help navigation.
- For disabled buttons and toggles when the IMINACTIVE is not defined by IMAGE is defined, we replace the non transparent colors by a darker version of the background color creating the disabled effect.
- New key K_PAUSE.

Windows

- Fixed dynamic cursor creation.
- Toggle with inactive image could be enabled/disabled only once.
- Fixed toggle in Radio behavior.
- Some keys were not being treated correctly.
- Improved key codes management.

Motif

- Fixed IupList setattribute VALUE and list items activated the ACTION callback.

Controls

- Circular IupDial now uses abssolute angle.
- CARET did not work when set inside EDITION_CB in IupMatrix.
- Check for double initialization of IupControls.
- Better resize management for IupVal and IupDial.
- IupControls now depends on the CD library version 4.3.3 in Motif.

IupLua

- Wrong implementation of DROPCHECK_CB.

Version 2.2 (11/Aug/2004)

INCOMPATIBILITIES

- Definition of K_parenleft changed to K_parentleft in C and all Lua bindings.
- Major IupLua5 change (see IupLua section below).
- IupLua4 is not supported.
- Motif 1.x is not supported.

General

- Documentation in Portuguese removed from the manual.
- Changed and documented the default palette used in IupImage.
- IupImage can now have up to 256 colors.
- New mouse wheel callback "WHEEL_CB" for Windows and Motif. If not defined the wheel will automatically scroll the canvas vertically.
- Changes on global attributes:
  "COMPUTERNAME", "USERNAME" - now implemented also in Motif.
  "COPYRIGHT" - not documented
  "SCREENDEPTH", "SYSTEMVERSION" - new for Windows and Motif
  "SYSTEM" - Implementation were different from the documentation
  "CURSORPOS" was documented as if it was only for Windows.
  "LOCKLOOP" now implemented also in Motif..
- The definitions IUP_SBDRAGV and IUP_SBDRAGH were not documented.
- Callback MENUSELECT_CB changed to HIGHLIGHT_CB. Now implemented also in Motif.
- New menu callback MENUCLOSE_CB.
- New utility functions IupMessagef and IupGetInt2.
- Improved visual appearance of IupScanf, IupAlarm and IupListDialog.
- New creation attribute "SEPARATOR" for IupLabel so you can create vertical or horizontal line separators.
- New IupGetText predefined dialog.
- Now all the predefined dialogs consult the global attribute IUP_PARENTDIALOG.
- New "HELP_CB" callback for all interactive controls.
- The "KEYPRESS_CB" callback now will be called repeatedly if the key is pressed and held.
- IupList can now have an edit box associated.
- The OLD newfocus parameter of the KILLFOCUS_CB is now NULL always, in Windows and Motif.
- The BGCOLOR color for IupImage transparency was not according to the documentation.
  It was using the default background color of the dialog.
  Now it uses the BGCOLOR of the control where it is inserted.

Windows

- Menus for notification icons (system tray) were not working correctly.
- Cursors in Windows now accept more than 2 colors and can have size different from 32x32.
- IupImage was rewritten in Windows to be more simple and flexible. This also solved some weird button backgrounds in gcc3.
- New global attributes "SHIFTKEY" and "CONTROLKEY" can be "ON" or "OFF", return the the key state (windows only).
- The default size for buttons in Windows was increased by 2 characters.
- Returning IUP_CLOSE in a SHOW_CB of an IupPopup wasn't closing dialog.
- IupOpen instead of initializing OLE, now only initializes COM (CoInitialize).
- The border of buttons are now drawn by a system function instead of simulated.
- New attribute "PLACEMENT" to show the dialog maximized or minimized.
- In IupFileDlg when browsing for folder it will use a new interface, with a resizable dialog and other features.
  Also in IupFileDlg fixed start position for IupPopup. New file selection callback and preview area. IupFileDlg was not using the IUP_PARENTDIALOG attribute. Default value for IUP_NOOVERWRITEPROMPT was wrong. ALLOW_NEW was inconsistent with the documentation.
- The button callback now is called only when the button is released inside the button area.
- WOM callback renamed to WOM_CB.
- New "HELPBUTTON" attribute for the dialog.
- The menu item now accepts auxiliary bitmaps.
- When the dialog has a multiline and the user press ESC the window was improperly closed.
- Fixed comboox resize feedback. When resizing the dialog the combobox was temporarily opened.
- IupCanvas was not receiving arrow keys events correctly in keypress_cb.
- IupHide now can close popup dialogs.
- Attribute TABSIZE for IupMultiline in Windows was not documented.
- Default value for attribute BGCOLOR for IupCanvas in Windows was not documented.
- Direction keys now are processed by the ACTION callback for IupText.
- The GETFOCUS_CB and KILLFOCUS_CB management for the controls was reviewed and optimized.
  GETFOCUS_CB now works for toggle and button.
- First RESIZE_CB of the canvas received a wrong canvas size.
- Label alignment for images was always center.

Motif

- New global attribute: "MOTIFVERSION".
- IUP_SBDRAGV and IUP_SBDRAGH were not implemented.
- HIGHLIGHT_CB menu item callback.
- "COMPUTERNAME", "USERNAME" and "LOCKLOOP" global attributes.
- IupMessage now uses native XmMessageBox.
- The overwrite confirmation dialog was closing the file open if the user answered "No".
- Implemented the IUP_NOOVERWRITEPROMPT attribute for IupFileDlg.
- The dropdown list now uses the Motif 2 combobox widget. So IUP is not compatible with Motif 1.x anymore.
- Now the GETFOCUS callback is also invoked when the list is dropdown.
- KEYPRESS_CB is now called only for IupCanvas.

Controls

- DEFAULTESC and DEFAULTENTER were missing in IupGetColor.
- New function IupLoadImage that uses the library IM to load an image file (implemented in an additional library).
- New dialog IupGetParam, similar to IupScanf but uses variable controls for fields.
- IupTabs now uses the FGCOLOR for the text color.
- ICTL_DASHED was missing in the documentation of IupGauge.
  The control now has the attributes MIN and MAX just like the valuator.
- For IupVal and IupDial, new keyboard and mouse wheel support.
  New attribute "SHOWTICKS" to show tick marks around the valuator.
  New attribute "UNIT" to change the angle unit to degrees in the dial.
  Completely changed visual of the controls.
  The controls can now be deactivated and it displays focus feedback.
- Updated visual for the IupGauge and IupTabs controls.
- In IupTabs the popup menu to select a tab sometimes did not set the new tab.

Matrix

- Documentation reviewed and reorganized.
- Returning IUP_CLOSE in CLICK_CB was not closing application.
- The scrollbar drag will now simultaneously scroll the matrix.
- New callback "DROPCHECK_CB" to aid the dropdown feedback in the cell.
- New utility functions: IupMatSetAttribute, IupMatStoreAttribute IupMatGetFloat, IupMatSetfAttribute, IupMatGetAttribute, IupMatGetInt.
- Fixed some display erros in Windows because of an error in the size of the scrollbar.
- In Windows pressing a key in a menu activates the k_any of the last active element. In the matrix this turns into an infinit loop. The matrix now uses the keypress_cb instead of the k_any callback.
- Fixed empty selection in the dropdown list if the user press a regular key to start editing the cell.
- Fixed invalid dropdown value if the user changed focus to the scrollbars.
- CLICK_CB was called twice in a double click (press+release).
- In Motif, the textbox and the dropdown did not open when you double click a cell. But now the user still needs to click again in the control to put it into focus.
- After editing the cell in the last line, now the focus goes to the column on the right at the last line, instead of the first line.
- BGCOLOR now works also for titles.
- FONT attribute now can be set/get just like BGCOLOR and FGCOLOR. But the cell size is calculated always from the matrix attribute IUP_FONT.

Tree

- Documentation reviewed and reorganized.
- CTRL and SHIFT accepts only values IUP_YES and IUP_NO.
  Default value of SHIFT and CONTROL is NO, it was NULL.
- Pressing Space without Control now activates the RENAMENODE_CB callback.

IupLua

- The selection callback wasn't working in Lua 5 binding.
- MOUSEMOVE_CB in Dial control was receiving wrong angle parameter in Lua 5 binding.
- IupGLCanvas wasn't working in Lua 5 binding.
- Major IupLua5 change.
  It now complies to LTN7 (namespaces). All exported functions are accessed only through **iup.FunctionName** (no Iup prefix anymore)
  All callbacks in Lua are now access through their exact name in the C API. Mostly add sufix "_cb" to name (most common callbacks renamed for ex: getfocus_cb, killfocus_cb). Also some names were fix: valuecb >> value_cb and mapcb >> map_cb.
  Numeric definitions also changed: IUP_DEFAULT >> iup.DEFAULT
  String definitions for values are no longer supported, use "YES", "NO", etc.
  iupcb changed to iup.colorbrowser.
- Use LoadLibrary to load IUP from Lua.
- There was no stack pop in color processing loop fo IupImage in IupLua5.
- IupLua4 is not supported anymore.

LEDC

- Added support for IupTree and IupSbox.
- Fixed include for IupColorBrowser.
- Fixed small invalid memory access.

Version 2.1 (18/Feb/2004)

General

- New split-panel control: IupSbox
- IupTree and IupMatrix libraries are now part of iupcontrols
- New functions to traverse IUP controls: IupGetNextChild, IupGetBrother, IupGetParent
- IupAppend accepts elements other than predefined internal controls (allowing CPI containers)
- Focus now may go to CPI controls
- Attribute IUP_X, IUP_Y are now valid for every control that has a native representation (returns the position of the control in screen coordinates)
- CURSORPOS global attribute is now returned from the driver
- IupGetFile was not allowing new files and should not change user directories
- IupGetFile was not accepting long directories
- IupAlarm does not take [ENTER] as button1 click anymore
- IupScanf does not accept "," when option is float
- Windows 95 is no longer supported

IupTree

- Trying to get attribute NAME for and invalid ID returns NULL
- Fixed attributes IUP_CTRL e IUP_SHIFT for mouse interaction

IupMatrix

- Special keys such as backspace, control+c, etc. are now ignored when not in edit mode
- leaveitem/enteritem were not being generated when the focus was leaving or entering the matrix
- leaveitem/enteritem should not being called when the cell enters edition mode through the mouse

Windows

- IupOpen/IupClose now initializes OLE (OleInitialize/OleUninitialize)
- ENTERWINDOW/LEAVEWINDOW reimplementation. LEAVEWINDOW does not fail anymore
- Mouse hook removed. Better performace
- New attributes TRAY, TRAYTIP and TRAYIMAGE and new callback TRAYCLICK_CB which allows a dialog to be put in the tray
- Action in IupText now responds to the [ENTER] key
  Some keys were not working with keypress callback: \ ] [' ; / . ,
- New attribute NATIVEPARENT, which makes any dialog in Windows able to be parent of a IUP dialog (even from other toolkits)
- Better protection dealing with other processes messages
- IupFileDialog when used to get directory was not updating STATUS attribute correctly
- IUP_APPEND small memory problem fix
- atexit removed
- KILLFOCUS_CB and GETFOCUS_CB were not being called when focus goes to the menu
- MAP_CB in a canvas is now called before RESIZE_CB (like the Motif driver)
- ALT-F4 was not working to close application
- Images sometimes show black using Visual C: do not use option in Visual C 6.0 /NODEFAULTLIB:libcd
- IUP_TIP does not show when the fade effect is on: MS fixed the problem, use autoupdate

IupLua 3.2, 4.0, 5.0

- Functions exported to Lua: IupGetType, IupGetParent, IupGetNextChild, IupGetBrother
- IupTimer, IupSbox binding
- IupTreeGetTable, IupTreeSetTableId, IupTreeGetTableId functions created
- Several bug fixes in IupLua 5.0
- New function iuplua_pushihandle, iuplua_dofile and iuplua_dostring, IupGetFromC
- If iuplua_dofile and iuplua_dostring are used errors are reported through _ERRORMESSAGE function
- Default _ERRORMESSAGE function shows a dialog with the error
- IupLua5: Removed Lua redefinitions of dofile and dostring
- Minor bug in IupTree function TreeSetValue
- IupListDialog was not returning a table as it should when in multiple mode

IupVal

- Attribute IUP_VALUE wasn't taking effect when set before mapping
- CD canvas was being altered during mouse movement event

Manual

- CPI manual revision
- IupLua manual revision
- Several examples revised
- Controls section rearranged

Distribution

- README on how to compile IUP with tecmake

Version 2.0.1 (31/Jul/2003)

General

- Attribute IUP_TYPENAME replaced by IupGetType function
- minor bugs introduced in 2.0 because of internal old misuse of the hash table.
- Following controls were not working with LED: val, dial, gl, matrix, tree.
- New canvas attribute "DRAWSIZE" that returns the drawing area of the canvas (in Windows we may have an addicional border included in "RASTERSIZE").

Windows

- Memory invasion when eliminating an item from an IupList with multiple items.
- Callback IUP_OPEN_CB sometimes was not being called.
- New dialog attribute "BRINGFRONT" which forces dialog to be the window in the front. Useful for multithreaded applications.
- Attribute ACTIVE was not working with radio control.
- Now folder selection in IupFileDlg uses IUP_DIRECTORY as a start path.
- Now when ESC or ENTER is pressed KEYPRESS_CB is generated

Motif

- Dropdown were becoming unstable when VALUE attribute is set after IupMap.
- Dropdown were not being positioned accordingly.

- IupList was not selecting the first item.
- IupTimer callback were called only once.
- The value "BGCOLOR" in a value of an image color table index appeared with erroneous color.
- keyboard and mouse callbacks were not being called when in full screen.

LEDC

- Updated to reflect 2.0 changes like "iupmatrx" to "iupmatrix".
- Now tests if name is not NULL before using IupSetHandle.

IupLua

- New binding for Lua 5. This is beta version since uses old notation "iuplabel" instead of "iup.label".

Version 2.0 (23/Jun/2003)

General

- IUP has undergone a large internal reorganization, but no structural or algorithmic changes have occurred. The purpose of this reorganization was to standardize function, variable and module nomenclature. This process is not yet complete, but the few remaining details will be solved in the next version.
- Table Hash was completely replaced with a modified version of Lua 4. This version is internal of IUP and does not affect applications. This has brought us a better management of the memory used by attributes.
- The CPI was changed to allow the creation of native controls, as well as controls based on IupCanvas. The internal controls were not yet rewritten over the new CPI - this will be done progressively in the next versions.
- The Ihandle definition changed from "void" to "typedef struct Ihandle_ Ihandle;". This has direct implications on C++ applications that did not do pointer typecast. In C++, code errors might occur and, in C, there might be warnings.
- New control IupTimer. Allows creating timers in Windows and Motif.
- New callback "KEYPRESS_CB". Allows intercepting any key and replacing all callbacks "K_xxx".
- IupHelp was rewritten in a simpler way. In Windows, it simply uses the system's configuration to open a URL and, in UNIX, it directly runs Netscape or another executable configured by an environment variable.
- New attribute "FULLSCREEN", allows creating a dialog that occupies exactly the whole screen.
- Dialog IupGetFile was rewritten using IupFileDlg.

Windows

- New attribute "CURSORPOS", allows programmatically changing the cursor's position on the screen.
- New attribute "NOOVERWRITEPROMPT" for IupFileDlg. It prevents IupFileDlg in Save mode from asking the user if s/he really wishes to overwrite a file.
- Problem corrected in the file list in the use of attribute "MULTIPLE_FILES" for IupFileDlg. When only a folder was selected, it was not setting the "STATUS" attribute in a cancelled action.
- Greater driver stability - Ihandle is no longer dependant on the native handle (HWND).
- New global attributes "HINSTANCE", "SYSTEMLANGUAGE", "COMPUTERNAME", "USERNAME".
- Global attribute IUP_SYSTEM now returns a more complete string.
- Cursor now changes instantly - it only changed before returning to IUP.
- In an inactive IupToggle, the IMINACTIVE image is now correct.

Motif

- The iupmot library no longer exists. Tecmake has been updated, but those who use their own metafiles must remove this file from the list of libraries in the application.
- New attribute "AUTOREPEAT" allows turning on and off the automatic repetition mode of pressed keys.

IupLua

- [4/5] IupListDialog when selection type is 1 (single) was not returning any value.
- [4/5] Callbacks mapcb and showcb had their names wrong: map_cb and show_cb
- [3] Callback action in IupMultiline was not passing the parameter "after".
- [4/5] In IupTree, callbacks "afterselection" and "beforeselection" were replaced with the callback "selection".

IupControls

- We have joined seven libraries in one: dial, gauge, cb, gc, mask, tabs and val. But neither the initialization functions nor each control's inclusion files were changed. The source code does not need to be altered, except for the makefiles. Tecmake was given a flag USE_IUPCONTROLS to automatically include this library.

IupMatrix

- The name of the library was changed from "iupmatrx" to "iupmatrix". The same for the inclusion files. Therefore, all applications that use IupMatrix must change the source code and the makefile to reflect these changes.

IupTree

- In one case, the active CD canvas was not being returned to the old canvas before drawing.

IupGL

- In Linux, the additional GLw library was added to the control library.
- New attributes for query in UNIX: CONTEXT (GLXContext), VISUAL (XVisualInfo*) , COLORMAP (Colormap).

History of Version 1.x

History of Changes in Version 1.x

Version 1.9.1 (17/Oct/2002)

General

- Version number now resides in iup.h (it is also included in the library during compilation.)

Windows

- IupLabel with \n was not working.
- Line-break in attribute IUP_TIP is now accepted.
- Double-click in the Windows top-left corner made the program crash.
- IUP_READONLY was only accepted if used before IupMap in a IupText or IupMultiline.
- Windows was limiting initial elements of a IupList to 999.
- New attribute FULLSCREEN created.
- The codes of the numeric keyboard when the CapsLock was turned on were not mapped correctly to IUP.
- New callback added MENUSELECT_CB (called when the mouse hovers over a menu or item.) - not fully implemented.
- Fixed IupList ACTION callback calls for pre-selected items on the first selection change.

Motif

- IUP_MOTFONT did not accept IUP fonts. Now it accepts both native fonts and IUP fonts.
- It is acceptable now to select an option in a popup menu with any mouse key.
- Attribute IUP_STATUS in a filedlg was not working in a silicon.

IupLua

- Better error messages.
- In the iuptree control, the callback BRANCHOPEN_CB was not passing the node parameter.
- In the iuptree control, new functions were implemented to associate and retrieve a Lua Table from a node or leaf.
- IupGLCanvas binding.

IupTree

- Expand and collapse no more alters selection of elements.
- When all nodes were deleted using "DELNODE0", "CHILDREN" inside a tree_selection callback, the program crashed.
- BRANCH_OPEN now passes parameter node.
- IUP_DEPTH now works for folders and leaves. Attention: the depth works only with the appointed element, not with its children.
- Some conditions necessary for a DEPTH change were wrong.
- Redraw optimization.
- When a tree was big, the scrollback was not working properly.
- When the tree was totally expanded and the scrollbar was all down, collapsing folders made the thumb be wrongly calculated.
- PGDN and PGUP were stopping in any folder that was closed.
- Even when the user did not want a folder or leaf to be selected, sometimes the tree allowed it.
- When the tree's folder does not have children, an empty box is shown next to it (instead of the + and - symbol.)
- Sometimes an error occurred in selection when a double click was done in a tree.
- Callback RENAMENODE_CB now works correctly.
- When the TreeSetValue function was used to define a tree, using a folder with no leaves made the program crash.
- New attribute "COLORid" allows the text color to be changed.

IupTabs

- **IUP_REPAINT was not repainting the elements in its interior.**

IupMatrx

- The attributes IUP_DEFAULTESC and IUP_DEFAULTENTER of a dialog were not working in Windows (they work only when the matrix is not in edition mode.)
- The matrix did not show the selected elements when the focus passed to another interface element.
- In a dropdown, when the user left edition mode changing the focus away from the matrix, the previously entered value was lost.
- Selection with the control key now works for selecting and deselecting.
- The cell with the input focus now draws the selection status.
- The attribute IUP_MARKED now works after the matrix is mapped.
- The matrix now starts with no cell selected.
- Clicking on the first column of a marked line with MARK_MODE LIN now also deselects the line.
- When MARK_MODE is LIN, COL or LINCOL the selection is not done on the focused cell.
- When MARK_MODE is CELL and MULTIPLE is NO the whole line cannot be marked.
- When MARK_MODE is NO nothing can be selected.
- The [TAB] key in the matrix now changes focus to next element.
- When MARK_MODE was NO (default), after leaving the edition mode with [ENTER] the cell was being marked.

IupVal

- Mousemove is now standardized.
- Idle is not used anymore (better optimization and code simplicity.)
- Minimum and maximum value when different from 0 and 1 now work.
- Clicking a position in the middle of the IupVal now work correctly.

Version 1.9.0 (18 Dec 2001)

General

- The K_ANY callback now considers the state of the CAPSLOCK key. The native behavior of the combination of the keys CAPSLOCK and SHIFT was kept.
- New binding for IUP: Lua 4.0.
- New binding for IupMask.

Windows

- Driver Windows now deals only with messages generated for IUP elements (this used to be a problem with CD's print dialog).
- Label fonts did not work when set before IupMap.
- Attribute IUP_FILTERUSED now can be set on before the creation of IupFileDlg.
- Tip in Windows now accepts \n.
- Tip in Windows is now modified immediatly after it is set though programming.
- Tip now can be removed immediatly.
- In a SubMenu, the attribute ACTIVE was not working propertly.
- The OPEN_CB callback was implemented in the SubMenu.

Motif

- Callback OPEN_CB in a SubMenu was providing wrong parameter.
- Attribute IUP_BORDER in a dialog was working differently from the manual when the window manager was sawfish.

iupMask

- iupMask was becoming unstable when the user set the attribute IUP_SELECTION in a IupText.
- There was a bug in the IupMask-IupMatrix combination.

IupMatrx

- Adding a new column or line is now correctly dealing with color inheritance.
- There was IUP_MARK_MODE defined but not: IUP_LIN, IUP_COL, IUP_LINCOL and IUP_CELL.
- The drop_cb callback was being called for any focus change. It is now being called just when the matrix enters edition mode.
- The matrix was not showing the selected cells when the user changed focus from the matrix.
- The matrix was not calling K_ANY from the parent if the callback had been set after matrix creation.
- IUP_RIGHTCLICK_CB is now called IUP_CLICK_CB. This callback is now called for every mouse button.
- New callback IUP_MOUSEMOVE_CB.

IupTree

- Attribute IUP_MARKED now also sets.
- IupTree's binding now exports functions to set and get ID.
- Redraw is now done with one attribute. This avoids unecessary redraw when the user wants to insert a lot of data.
- IupTree now takes leafs and nodes before IupMap.
- Clicking to select a LEAF was not always working in Windows.
- BRANCHOPEN and BRANCHCLOSE callbacks were not testing the return value correctly.
- Double clicking was not working propertly. When the user clicked a node, while the timer was still waiting for the second click, it was impossible to click a nother node.
- Hitting the space button with CTRL pressed now marks the element immediatly.
- SELECTION_CB callback was created. This callback is called when any type of mark is made on the Tree. The return value blocks this action.
- Removed callbacks BEFORESELECTION_CB and AFTERSELECTION_CB.
- Setting IUP_VALUE though programming does not activate callbacks anymore.
- Keyboard control, including arrow keys, PGUP, PGDOWN, HOME e END were not working propertly.
- Clicking + or - was not activating the SELECTION_CB callback.
- SELECTION_CB is now in the binding. BEFORESELECTION_CB and AFTERSELECTION_CB are not.
- The IUP_MARKEDid attribute now returns IUP_YES or IUP_NO depending on the state of the node's mark. If the node does not exist, the returned value is NULL.
- IupTree was breaking when it tried to erase a marked node inside BRANCHCLOSE_CB.
- The BRANCHCLOSE_CB callback was not being called for the correct node.
- SELECTION_CB was included in the binding.
- Including a new leaf now does not alter selection.

IupGL

- Created attribute "ERROR" indicating error in a GL canvas.

IupCB

- User canvas was not being reactivated after the mouse callbacks.

IupLua

- IupGetGlobal and IupSetGlobal were not doing toupper.
- New function created to get an Ihandle created in C: IupGetFromC.
- The IUP_BUTTON_CB callback was not being called.
- Functions isshift, iscontrol, isbutton1, isbutton2, isbutton3 and isdouble are now exported.
- IupPreviousField and IupNextField were not implemented.
- The OPEN_CB callback was implemented in the binding with the name OPEN.
- New callback IUP_MOUSEMOVE_CB for matrix.

Version 1.8.9 (07 May 2001)

IupMatrx Control

- If the user defined FGCOLOR while the matrix was in edition mode, the application crashed.
- Hitting Esc was causing garbage to be written in the matrix field.
- A bug that made the value_edit callback be called several times was fixed (it was called several times because the matrix kept trying to exit the edition mode with other events).

IupTree Control

- New IupTree control.
- Scrollbar.
- Multiple selection.
- Default image size: 16x16.
- Lua Binding.

IupCB Control

- The name of the Lua colorbrowser element has changed. Now it is called iupcb, not cb.

Windows

- The IUP_MULTIPLEFILES attribute was created. Now it is possible, in Windows, to select several files in a FileDlg.
- IupHelp now only initializes DDE when it is used.

Version 1.8.8 (15 Mar 2001)

- The global.h, macros.h, rgb.h and hls.h files are no longer exported by IUP.
- Some keys were in conflict among themselves (shift-home and 4, for instance). Shift-space and Ctrl-space were added to the K_ANY callback (Windows and Motif).
- IUP_VISIBLE was returning NULL on IUP when the dialog was not mapped.
- IupSetLanguage can now be called before IupOpen();
- iuptoolbar and iupfiletext were removed from the distribution.

CPI

- Several defines (such as strieq) are no longer exported from iupcpi.h
- Functions iupAddSymbol, iupGetSymbol, iupgetdata and iupsetdata are no longer exported from the CPI.

Motif

- The Tip font is now inherited from the element it belongs.
- Inserting a text (IUP_INSERT or IUP_APPEND) on Motif was ignoring the maximum number of characters.
- Some ITALIC fonts were not working.
- Several visibility problems were fixed for ZBOX inside a ZBOX.
- The default value of the ALLOWNEW attribute (in fileopen mode) allowed creating a new file (now standardized).

IupTabs Control

- IupTabs was not considering attribute IUP_ALIGNMENT.
- Tabs was not showing the selected element if it was selected while the Tabs was invisible (it was a Motif bug).
- The <TAB> key was neither passing the focus to IupTabs nor taking the focus off it.
- The SIZE attribute is now defined for the tabs of IupTabs ICTL_TABSIZE.
- Changing the text value for Tabs was not recomputing the Tabs size.
- The appearance of IupTabs was enhanced.
- IupTabs now sends the focus back to the first element when the user tries to shift right after the last element.
- Now a redraw can be forced on Tabs with the IUP_REDRAW attribute.

IupMatrx Control

- Ctrl+arrows was not working properly.
- The behavior of the DEL key to delete a set of cells now also considers the return of the IUP_EDITION_CB callback.

- The mark is now shown (not the focus) when matrx loses the focus (users were having problems when wishing to hit a button to cause an action over the matrix).
- Oh the NT platform, the fields of the created matrix had the wrong values when an automatic scroll occurred.
- Right-clicking the matrix now passes the control parameter (as in BUTTON_CB) isshift(r), iscontrol(r), isbutton1(r), isbutton2(r), isbutton3(r), isdouble(r)
- Vertically scrolling by dragging the thumb now works properly.
- The focus is now correctly drawn inside the matrix (when only half the cell appears, half of the focus is drawn).
- When leaving the edition mode by clicking an element outside the matrix, the focus was remaining on the IupText in the matrix.
- Colors and alignments are now moved when a cell is moved either by adding new lines or columns or by deleting lines or columns.
- The matrix now leaves the edition mode whenever lines or columns are removed.
- When the user clicked a cell near the end of the matrix (on the x coordinate) an automatic scroll was made and the cell beside the desired cell was marked.

Windows

- KEY in IupItem was replicating the underlined KEYs (and some times adding the wrong values because of that).

IupLua.exe

- Now works properly with all controls.

IUP Manual

- All elements now have examples at least in IupLua and C.
- The IupMask manual was created.

Version 1.8.7 (23 Nov 2000)

- The alignment of composition elements can now be changed on-the-fly.
- Current language treatment has been changed. ATTENTION: previous putenv no longer works! Use new functions IupSetLanguage and IupGetLanguage. Default language: Portuguese.
- IupAlarm's design was reformulated. Now all buttons have the same size.
- Functions IupUnMapFont and IupMapFont were created to make the use of the drivers fonts easier.
- Attribute IUP_FONT now accepts a string either with the native font or the IUP font, and always returns the native font (attributes WINFONT and MOTFONT are now obsolete).

Motif

- Motif did not have K_ANY for IupList in dropdown mode.
- The IUP_VISIBLE attribute now works for FRAME, ZBOX, VBOX, HBOX and RADIO (all elements were tested). Now it is no longer lost for internal HBOX elements when the HBOX visibility is changed.
- When the user changed from one ZBOX to another, the first one was forgetting which elements were visible.

Windows

- When Toggle 1 (default) begins deactivated, it no longer remains marked forever.
- Toggle with image now accepts images IUP_IMPRESS and IUP_IMINACTIVE, but it follows the Windows standard for Toggle manipulation.
- Toggle was not verifying whether it was active or not when it was created.
- Canvas redraw was optimized. The canvas now uses transparent color as default. The user is in charge of drawing the canvas, but now it no longer blinks when a redraw is made. Tip: To avoid unnecessary canvas redraws, do not put it inside a frame and use the IUP_CLIPCHILDREN attribute.
- Initializing Toggle (or Radio) with a value and then modifying it via callback was marking both toggles.
- Changing Toggles color (IUP_FGCOLOR) was not working on Windows unless its background color was also changed.
- IupItem outside a submenu was not calling the callback.
- On Windows, the IUP_HOTSPOT attribute was being read incorrectly (the correct form is with ":").

IupMatrix Control

- DROPDOWNs function in Matrix was corrected. Now the user fulfills the dropdown values, which always start at position 1. If the user wishes, he/she can set the initial dropdown value by checking the IUP_PREVIOUSVALUE attribute about the dropdown element passed as parameter. This attribute returns the previously selected string value.
- Dropdown now enters edition mode just as regular fields do.
- Dropdown can automatically close after the users choice. Simply return IUP_CONTINUE for the callback chosen by the dropdown.
- Now the dropdown accepts the ESC key, restoring its previous value.
- An element with focus is now drawn with double focus.
- The color of a selected element is now 20% attenuated.
- When the user entered edition mode using the mouse and exited it hitting ENTER, the cell remained selected.
- Matrix no longer gets lost when it has 0 lines.
- Matrix was not accepting a user to return a constant string with \n from a callback.
- A Matrix that loses the focus does not lose the selection (but it is not apparent).
- TAB no longer changes cells in the Matrix (it now changes IUP elements).
- Hitting delete on a marked element deletes everything.
- Matrix leaves the edition mode when IupTexts exit arrows are used.
- There was a computation mistake in cell size when the Matrix was in edition mode.
- When the user scrolls, the Matrix exits the edition mode.
- ALL problems caused by cdActivate in Matrix were solved.

Other Extended Controls

- The element from IupGL was not getting the focus when it was the only element in the dialog.
- In IupGL, OpenGL now synchronizes its functioning with Motif (glXWaitX) at resize.
- IupGC now works with IUP_ENGLISHs variable set (cancel/cancela, red/Verm, etc.)
- IupGauge now accepts changing text or percentage values on-the-fly.
- Tabs font now has a differentiated color when it is inactive.

IupLua

- IupScanf at IupLua was not performing the final dialogs popup.
- IupSetLanguage, IupGetLanguage, IupMapFont and IupUnMapFont were created at IupLua.
- It now considers the IUPLUA_QUIET attribute.
- The callbacks in IupLua are now inherited (eg.: k_any from a dialog is called when IupCanvas does not have k_any).
- The librarys opening message now follows a standard.
- IupLua was passing Luas pointer to IUP instead of copying its value in IupSetHandle (making it crash).

IupLua Program

- iuplua was not running with IupVal and IupGetColor.
- iuplua now accepts several files as a parameter.
- iuplua is now joined with iupluafull
- iuplua now shows line number and cursor column.

Version 1.8.6 (21 Jun 2000)

- All libraries were generated for AIX 4.3.2, which is available in new IBM machines.
- A series of memory management problems was solved for all platforms.
- Attribute IUP_SELECTEDTEXT now can also be used to change the selected text in a IupText and IupMultiline field.
- The IupLabel element now takes the IUP_ALIGNMENT attribute into account.

- The IupList (dropdown) element now always leaves some option selected (unless there is none to select).
- When the selected elements value in IupList (dropdown) is changed, it now remains selected with the new value.

User Manual

- The user manual is now also available in several Windows Help formats, including the help format for Visual C++ (5 and 6). To configure your account for Visual C++ to access IUPs Help, run W:\iup\help\iuphelp.reg (ATTENTION: On Visual Studio, IUPs manual must be activated and deactivated through option Help -> Use extension Help). Other available formats can be found at W:\iup\Help.
- A general revision of the user manual is being made.
- The CPI manual was rewritten.
- Several examples were included.
- An application called iupluatest (W:\iup\bin) was created to run the IupLua examples included in the manual (it works with the controls using the installed DLLs).

Windows

- There is no longer any restriction for the number of dialogs created using IUP (the only limitation now is Windows capacity to create native elements).
- Events of IupButton and IupToggle were being improperly called when a IupHide or a IupShow was made on the dialog.
- A bug when drawing an image associated to a IupToggle element was fixed.
- The functioning of attributes IUP_DEFAULTENTER and IUP_DEFAULTESC was corrected.
- Now, when a user changes the selection of a multiple IupList via programming, IUP internally updates the selection.
- The IUP_BGCOLOR attribute to define a new cursor was not standardized with the Motif, and color 0 in the Windows image was never allowed to be transparent.
- A bug in the dropdown list was fixed. It was not calling callback GETFOCUS_CB, causing instability in the IupMatrix element).
- The transparency color in a cursor now must be color number 0 (according to the manual, this is the way it was supposed to be).
- The IupList (dropdown) callback is no longer called for element 0 (which does not exist).
- A button in a Popup dialog was only allowing to be pressed via mouse. Now it can be pressed with the space key.
- The IupSetAttribute(x,IUP_VISIBLE,IUP_YES) call, when x was a dialog, was not working.
- Calling IupHide with a frame, with [hvz]box or with radio was not the same thing as calling IupSetAttribute(n,IUP_VISIBLE,IUP_NO)".
- The IUP_MOUSEPOS position in a dialogs IupPopup was not functioning.

Motif

- Several memory leaks were fixed. They occurred when IupGetAttribute called functions from XM which allocated memory to store the attributes value. This change may cause problems for applications which did not copy the value returned from IupGetAttribute and used the returned string. This usage of the return value from IupGetAttribute is not appropriate, because the user has to copy this string if he/she intends to remain using it (the returned string is intern to IUP).
- The dialog's Close callback was not closing the application when it returned IUP_CLOSE.
- The IUP_ACTION callback from IupMultiline was not returning the new text value if the key was validated (parameter after).
- The dropdown list was not automatically showing the first element when it was opened.
- The Motif now returns the default font when IupGetAttribute(n,IUP_FONT) is performed.

IupLua

- The names of callbacks show_cb and map_cb were corrected.
- A bug that made a toggle image not appear was fixed.

Extended Controls

- The default cursor of the IupMatrix element now looks like the MS Excel cursor. (Remember to call IupMatrixOpen() even when using IupLua!)
- Alignment (center) of the field in column 0 of the IupMatrix element.
- The user can now return IUP_CONTINUE at the action callback of element IupMatrix to allow IUP to go on treating pressed keys in the conventional IUP way.
- The dropdown list at IupMatrix was losing its current value when the user changed cells.
- The IupGetColor element was being drawn outside the canvas (old problem in cdActivate).
- The font in IupTabs is now inherited.
- Attributes ICTL_ACTIVE_FONT, ICTL_INACTIVE_FONT, ICTL_FONT were implemented in the IupTabs element.
- Attribute IUP_MARGIN was implemented for the IupGauge element.

Version 1.8.5 (18 Apr 2000)

- The versions of libraries IUP and IupLua were synchronized. From this version on, these tools will be distributed together.
- The library generation mechanism was changed to use libmake. All DLLs are available and following the same standard as the DLLs of other Tecgraf libraries.
- A FAQ was created for IUP: http://www.tecgraf.puc-rio.br/~mark/iup/faq-iup.txt.
- Several memory management problems were fixed.
- Attribute IUP_DIALOGTYPE can now assume three values: IUP_OPEN, IUP_SAVE and IUP_DIR. Due to the creation of IUP_DIR, the IUP_ALLOWDIR attribute is no longer used.
- One more value was added to attribute BGCOLOR: IUP_TRANSPARENT (used only by the Canvas to avoid unnecessary drawing).
- Function IupGetError was removed from iup.h.
- Function IupDataEntry was removed from iup.h.

Windows

- Function iupdrvSetIdleFunction was added to make the Windows compatible with Motif.
- The bug that made IUP crash when using MessageBox inside a button callback was fixed.
- IupDestroy now reconfigures the button control function (it was making IUP crash).
- The IUP_READONLY attribute was implemented (valid for Text and Multiline).
- The IUP_FILTERUSED attribute was implemented: it informs which is the filter selected by the user (1, 2, 3...).
- A bug that caused IupPopup(IupMenu(item)) not to call the items callback was fixed.

Motif

- IupDestroy was corrected. In a IupFrame, it made IUP crash.
- IupList was corrected. It crashed when the user changed its elements and tried to set IUP_VALUE.
- The memory leak at IupGetFile was removed.
- List elements were not being correctly deleted.

IupMatrix Element

- The bug in the NT matrix was fixed. It was not refreshing added elements (the values on the cells were wrong).
- The bug in the scroll matrix was fixed.

Version 1.8.4 (09 Dec 1999)

Windows

- A problem, which called the dropdown callback even for an already-deleted element, was fixed.
- Function IupHelp is now available.
- A bug was fixed; it caused excessive system resource usage when dialogs with several elements were used.
- The size of the version dialog was corrected.
- A bug was fixed; it made IUP crash depending on the use of MessageBox. Same for IupFileDialog.
- Callback IUP_BUTTON_CB was added for the IupButton element.
- A bug was fixed; it made IupGetInt(d,IUP_X) return a wrong value when the dialog was maximized.

CPI Controls

- The color inheritance problem was fixed.
- Corrections were made to the Dial size.
- Attributes of colors FGCOLOR, BGCOLOR, and fonts FONT, WINFONT, MOTIFFONT.

Version 1.8.3 (15 Jun 1999)

Windows

- The IUP_ACTIVE attribute now also works in the frame.
- The action callback in Multiline now also accepts the DEL key.
- Toggle element now accepts an image.
- The IUP_TOOLBOX attribute was implemented for dialogs.
- A bug was removed; it made a second IupShow in a dialog reset its position to the center of the screen.
- Treatment of the SIZE and RASTERSIZE attributes was changed.
- The IUP_ACTION callback now treats the DEL key and commands and keys from the Cut and Paste menu.
- A conflict was solved; it made the key - generate a call to the callback as if it were key (plic).
- Keyboard accelerators for menus now work, since the focus is no longer on the dialog. When a dialog receives the focus back, it sets the focus to the last control inside it that had the focus.
- IUP_K_ANY no longer issues beeps when keys are pressed on the canvas.
- When the IUP_STARTFOCUS attribute is not defined, the focus is set for the first control in the dialog that accepts it, thus preventing the dialog from keeping the focus and allowing the menus to be called via accelerator.
- Attribute IUP_SELECTION was implemented.

Motif

- Color management for 8bpp displays (256 colors) was re-implemented. Basic colors used by IUP (black, white and the grays used for highlight and shadow) are now reserved, and the search for colors in the palette was optimized.
- Elements such as IupCanvas now have their own visual, independent from their parents. If allowed by the display, the default visual of a canvas will be TrueColor (24bpp); if not, it will be the same as the default display visual.
- The IupToggle element now processes the IMAGE attribute differently: it now shows the toggle with the same appearance as the IupButton element, but maintaining its functionality the button remains pressed until the user clicks it again. The IMPRESS attribute can be used to define the image used for the pressed button. In this case, the user is in charge of giving it a 3D appearance.
- IMPORTANT: The size of the dialog can be adjusted after being mapped, by means of the SIZE and RASTERSIZE attributes
  - The size of the dialog has now precedence over the smallest size required by its children (either having been specified in its creation or in run-time).
  - Attributing a NULL value to the SIZE or RASTERSIZE (in C) of a dialog will re-compute its size according to the size of its children.
  - Partial dimensions (###x and x###) are now treated correctly.
  - Therefore, applications that define sizes for dialogs (either in LED or in C) smaller than the minimum size required by their children will show truncated dialogs. To force a computation based on the size of the children, set any of these attributes to NULL (in C) or simply do not define them in LED. As a general rule, avoid specifying a dialog size unless there is a real need for such in this case, be careful to specify a sufficient size.
- IupFileDlg:
  - The default value for the DIALOGTYPE attribute was not being recognized (the program aborted when there was no defined value).
  - When ALLOWNEW = NO, the dialog informs if the user is specifying a non-existing file (instead of simply returning, as was happening).
  - When the dialog type was OPEN, the returned value was 1 (Cancel) even when the user confirmed the operation.
  - If DIALOGTYPE is SAVE, a confirmation is required if the file already exists.
  - A new dialog was created for each popup without destroying the previous dialog.
  - The NOCHANGEDIR attribute was implemented.
  - The dialog does not return if the user specifies a new file when attribute ALLOWNEW = NO. The same happens when attribute ALLOWDIR = NO and a directory is specified. In these cases, alerts are shown.
- The iupGetColor function for CPI controls was replaced in functionality by the iupGetRGB function (iupGetColor is maintained for compatibility purposes, but it should no longer be used).
- TRUECOLORCANVAS was created. It indicates if the display allows the creation of TrueColor windows (> 8bpp), even if the default is PseudoColor.
- Tabs: a problem was fixed concerning the use of the VISIBLE attribute for elements belonging to a non-selected tab.
- IupHelp: allows using a browser (default = Netscape) for viewing HTML pages.
- The ACTION_CB callback, from IupText, now receives, apart from Ihandle* and int, a char* pointing to the new text value in case the key is confirmed.
- Dropdown lists were not correctly processing the VISIBLE attribute.
- A problem with the initialization of multiple-selection lists was solved: the VALUE attribute was not being respected in some cases.
- Attributes FGCOLOR and BGCOLOR from the dropdown list were not being correctly updated.
- IupLoopStep was re-implemented: now it no longer blocks when there are no events to be processed (it simply returns DEFAULT).
- The dropdown list is closed when the associated textbox is totally or partially darkened.
- The dropdown list was not being closed when the dialog lost the focus if IupIdle was registered.
- A problem in the exhibition of CPI controls was fixed.
- New return code (CONTINUE) was created, specific for key callbacks, to be used when the event is to be propagated to the parent of the element receiving it.
- In some situations, elements destroyed by means of IupDestroy were receiving events, making the application abort.
- The redefinition of items in the main menu was making the dialog return to its original size.
- Consulting attribute BGCOLOR in a dropdown list was aborting the application.
- Consulting attributes BGCOLOR and FGCOLOR of a canvas with a different visual from the default was generating an X-Windows error message.
- The problem with IupFileDlg was fixed (the application was aborting).
- IupDestroy in a bar menu was inducting an infinite loop to the application.
- The list now matches the documentation: it calls the action callback for the de-selected element (with the v = 0 parameter).
- Bug correction: The use of a Motif attribute instead of a function was making Motif lost control of memory management (memory already liberated was liberated again, which aborted the application).
- ACTION in IupText caused SIGSEV when the user pressed ENTER.
- New IupMapFont for mapping IUP fonts -> Motif.

Version 1.8.2

Windows (12 Jan 99)

- Function char* IupMapFont(char* font) converts a IUP font describer (used by the IUP_FONT attribute) into a native font describer (used by IUP_WIN_FONT).
- File Drag & Drop was implemented in dialogs and canvases, via the IUP_DROPFILES_CB callback.
- Attribute IUP_EXTFILTER was implemented for the IupFileDlg control, allowing the use of more than one filter.
- Changes were made to allow the creation of CPI elements other than CANVASes or dialogs.
- The IUP_ACTIVE attribute of a dialog can now be changed after it was mapped.
- List callback correction: the callback is now called both for selected and not selected items.
- New function void IupHelp(char *url) shows a URL in a Netscape window.
- The treatment of the new return value for keyboard callbacks, IUP_CONTINUE, was implemented.
- IUP_CURSOR attribute was implemented.
- A code was added to treat the case of toggle de-selection via IupSetAttribute.
- IUP_CARET now uses , as a separator instead of old :.
- A restriction was eliminated that prevented the function iupGetTextSize from being called passing a dialog or frame as a parameter.
- New text callback was implemented; it receives the text both before and after the change, and receives the code of the typed key.
- It was possible to set two activated radio toggles by selecting VALUE for one of them on the radio and VALUE = ON on the other toggle.
- Attributes IUP_STARTFOCUS, IUP_DEFAULTENTER and IUP_DEFAULTESC were implemented.
- The IUP_VALUE of a IupRadio was not allowing to be changed if it was not visible.
- A problem was corrected for the lists, which were being reset between a IupShow/IupPopup and another.
- Attribute IUP_SELECTEDTEXT was implemented. It returns the selected text (if there is any), with the \r already filtered.
- A bug was corrected; it caused and Assertion Failed when the mouse was moved after a window was destroyed.
- The value of IUP_VALUE of a IupText and a IupMultiline now does not contain \r.

Motif v1.8.2 (14 Aug 98)

- IupFileDlg was corrected: the IUP_FILE and IUP_DIR attributes were not being treated correctly.

- In some specific situations, closing a dialog could lead to the end of IupMainLoop, causing an abortion of the application.

Version 1.8.1

Windows v1.8.1 (17 Jul 98)

- Correction: IUPs Matrix element was being shown with different fonts from the ones used by IUP, especially on UNIX platforms.
- A bug related to ZBOX was fixed.
- IupAppend on Multiline now includes \n at the end of the text.
- A font set by CD no longer affects canvas size computation.
- IupSetAttribute from a IupRadios VALUE with the name of a toggle with more than one name now works.
- Default attributes now store values that match the documentation.
- Function IupFlush was implemented.
- Small errors in dialog size computations were corrected.
- Now the dialog size is changed when the size of one of its children increases.

Motif v1.8.1 (16 Jun 98)

- Correction: IUPs Matrix element was being shown with different fonts from the ones used by IUP, especially on UNIX platforms.
- Dropdown list (combo box) remained opened if the element was hidden or destroyed.
- The use of popup dialogs was sometimes preventing the last IUP_CLOSE (or IUP_DEFAULT) from ending IupMainLoop.
- [LINUX] The button press event was not being received by the canvas when the CTRL key was pressed.

Version 1.8 (29 May 98)

General (also includes changes to both drivers)

- BUG: Valuator, Dial and Gauge could cause an invalid memory access on resize or destroy.
- BUG: The parse of CPI elements described in LED was corrected.
- BUG: Valuator was removing the applications idle action.
- NEW: FILEDLG control.
- NEW: IupStoreAttribute function.
- NEW: IupSetfAttribute function.
- NEW: IupSetGlobal, IupGetGlobal and IupStoreGlobal functions for global attributes.
- NEW: K_sCR key; shift-enter combination is now treated by IUP (callback: IUP_K_sCR, code: K_sCR).
- NEW: IUP_TYPENAME attribute returns the name of the element type.
- NEW: CPI popup method.
- NEW: Definition of global attributes (verification only) IUP_VERSION, IUP_DRIVER, IUP_SYSTEM and IUP_SCREENSIZE.
- NEW: Attributes IUP_X and IUP_Y were implemented, for dialogs only. They provide the dialogs upper left corner coordinates in relation to the upper left corner of the screen.
- NEW: IUP_SHRINK attribute to change the computation of the position and size of elements.
- NEW: CPI control for an OpenGL canvas.
- CHANGE: The IUP_TYPE attribute of the IupFileDlg control was changed into IUP_DIALOGTYPE, which must contain OPEN, SAVE or NULL.
- CHANGE: The IupSetAttributes function now returns the Ihandle*.
- CHANGE: The IupSetAttribute function no longer returns the old value.
- CHANGE: CPIs create method now creates the handle.
- CHANGE: New function for CPI class creation.
- CHANGE: Some obsolete definitions of iup.h are now only available when the IUP_COMPAT macro is set.
- CHANGE: The ICTL_TYPE attribute of the IupTabs control was changed to ICTL_TABTYPE.

Lua Binding

- NEW: iupkey_open function allows using IUPs key definitions in Lua.

Windows

- NEW: Image now accepts BGCOLOR color. This turns the color associated to the index into the background color of the element linked to the image.
- BUG: the IUP_TITLE attribute of the IupItem element can now be changed after the element has been mapped.
- BUG: A color problem was fixed; it occurred when the name or path of the executable file contained spaces.

Motif

- BUG: The dropdown list no longer remains on the screen.
- BUG: The computation of scrollbar attributes POSX and POSY was fixed.
- BUG: Double-click was only being generated for the first button.
- BUG: FRAME layout was corrected.
- BUG: The color of the menu item was corrected.
- BUG: The management of the nested elements of a ZBOX and/or with the VISIBLE attribute defined for its children was fixed.
- BUG: The color remained undefined when the value of attribute FGCOLOR or BGCOLOR was not valid.
- BUG: General cleaning was made to remove memory leaks from the driver.
- NEW: Attributes IUP_X and IUP_Y to provide the pixel position of any element.
- NEW: Attribute IUP_RASTERSIZE can be consulted.
- NEW: Menu item now accepts \t to align the text to the right Windows already allowed it.
- NEW: Version number was added; can be retrieved with tecver.
- CHANGE: Multilines scrollbar is no longer deactivated with ACTIVE=NO.
- CHANGE: Multilines and lists BGCOLOR no longer affects the scrollbars.

Version 1.7

- The implemented code was made compatible with manual specifications. iup.h was changed to reflect that. To use old definitions, set IUP_COMPAT before including the iup.h file to the applications.

To Do

Roadmap for the Next Versions

Version 4.0

- C++ API
- Lua script editor and interactive debugger
- Interactive Dialog Editor / Tools to improve productivity
- Complete the Tutorial
- Remove DEPRECATED functions

Next Versions (?)

- Other plot modes: STEP, STEM, CHART, BARHORIZ, MULTIBARS, ERRORBAR (**IupPlot**)
- **IupGLText** (using a dynamically displayed IupText when editing)
- **IupFlatTabs**

- Val, link and spin for cells in **IupMatrix**
- Colorbar in **IupPlot**
- Segment pick in **IupPlot**
- Merge cells support for **IupMatrix**
- Interactive change of column position in **IupMatrix**

Future Versions (?)

- Dialog Templates
- MacOS X native driver

General

- **Important:** remove DEPRECATED functions and headers in IUP 4.0
- **Important:** Interactive Dialog Editor
- **Important:** One big Controls Demo just like GTK, wxWidgets and Qt have.
- **Important:** RPM, Debian and LuaRocks distribution packages.
- **Important: IupGLCanvas** in MACOS X using native OpenGL support.
- **Important:** a MacOS X native driver using Carbon or Cocoa.
- **Important:** Lua script editor and interactive debugger
- Support for image and text at the same time in **IupLabel**.
- The actual model for control data structure in the internal SDK is restricted for derived classes.
- Loading and saving RTF files in **IupText**. Add support for images inside the text.
- Possibility to change the system menu in Windows. Support for cascading **IupPopup** for menus.
- Vertical text in labels and buttons.

Windows

- Known Issue: when in Windows 8 the **IupFontDlg** dialog does not supports the TITLE attribute nor can be positioned.
- Known Issue: when an **IupVal** is inside an **IupTabs**, the tabs disappear when the mouse moves over it after being used in the valuator. A workaround is to put the valuator inside an **IupFrame** and then inside the **IupTabs**, so the problem does not occur.
- Known Issue: when the dialog background is dynamically changed the **IupVal** background is only updated after the user click on the control or when the control is redisplayed.
- Known Issue: in Windows Vista the COMPOSITE=YES attribute of the **IupDialog** is not working as expected. There is still flicker when the dialog is resized. **IupTabs** in Windows Vista when COMPOSITE=YES works only if MULTILINE=YES. (since 3.0)
- Known Issue: in Windows when CANFOCUS=NO only the Tab key navigation is not done, when clicked the control will still get the focus. The only exceptions are button and canvas.
- Known Issue (Compiler): the **IupImgLib** takes an very long time to compile under Visual C++ up to version VC9 (starting in VC10 the problem does no occur). (since 3.0)
- Known Issue (Compiler): when building with Open Watcom the additional controls crash. When you add debug information to the main IUP library the problem solves. We tried to track down this error but it does not occurs with debug information and our attempts without debug does not gives any results. So the IUP main library for Watcom is now distributed with debug information. (since 3.0)

GTK

- **Important:** stop using deprecated functions in GTK 3.14. Must not define GTK_DISABLE_DEPRECATED in "iup/src/config.mak" to be able to compile IUP in Ubuntu 15.
- Known Issue: can not set focus to a child inside TABCHANGE_CB or TABCHANGEPOS_CB in **IupTabs**, because internally GTK will always set the focus to the first child.
- Known Issue: in Ubuntu 11.10 the canvas scrollbar is not notifying IUP that the user dragged the control. To solve the problem remove the overlay-scrollbar, this is the package that makes the scrollbar invisible until the mouse is near it.

Motif

- Known Issue: when the **IupList** has DROPDOWN=Yes in Motif, and the list has items with the same string, the ACTION callback will return the index of the item with the first instance of the string only. This seems to be a bug or limitation in Motif.
- Known Issue: an element when inside an **IupScrollBox** is not being displayed until the box is scrolled if its size alone is greater than the scrollbox visible size.
- Known Issue: **IupMatrix** crash the application during its creation on OpenMotif 2.3.3. inside the creation of the internal **IupList**.

Lua Binding

- **Important:** create a base library for exported functions. All other libraries will be pure Lua modules only.
- **Important:** remove the standard "lib" prefix from pure modules dynamic libraries names in UNIX.
- **Important:** remove the Lua version number suffix from all libraries.

IupMatrixEx

- **Important:** drag&drop of columns, i.e. interactive change of column position.

IupTree

- Known Issue: the rubber band gets lost depending on what you do inside the SELECTION_CB callback in GTK. To avoid that set RUBBERBAND=NO.
- Known Issue: the SELECTION_CB callback may be called more than once for the same node with the same status.
- Known Issue: in Windows XP, when using a font for an node with TITLEFONTid in **IupTree** that is larger than the element FONT the item text will be cropped at right and bottom because the system uses the element font to calculate the item size. The only exception is when you just change the font to add a Bold style.
- Important: old NAMEid attribute conflicts with the common attribute NAME. Should be replaced by the new TITLEid. NAMEid will be removed in future versions. (since 3.0)
- RENAMEEDIT_CB callback and RENAMEMASKid attribute.
- Add new nomenclature option for id, for example ":2:1:4"
- Define minimum size based on tree nodes.
- drag&drop of multiple selected nodes.

IupPPlot (DEPRECATED)

- Allow to select multiple nodes at once dragging the mouse within a region.
- Add support of legend text near each dataset plot.
- Adjust AutoScale to start and end at major axis ticks.
- Improve the display of values near each sample.
- Custom legend position.
- PPlot force the definitions of the margins. It should have a way to automatic calculate the margins when doing automatic scaling.
- PPlot force the Grid to be automatically spaced following the major ticks.
- PPlot generates lots of warnings on all compilers.

IupMglPlot

- **Important:** text render quality is lower than in **IupPlot**.
- **Important:** add UTF-8 mode using MathGL Unicode support.
- Compile MathGL using OpenMP support.
- Logarithm scale is not working properly.
- Automatic ticks computation needs to be improved.

Possible New Controls

- **CanvasCD** - an IupCanvas with a CD canvas associated.
- **ProgressIndicator** - a simpler version of the IupProgressBar

- **Calendar** (Windows, GTK)
- ----------
- DropDownButton - Mix between a drop down list and a button
- Table - similar to IupMatrix but using native controls (Windows,GTK,Motif)
- Scrollbar - just the scrollbar as a control. (Windows,GTK,Motif)
- PropertyGrid - a 2 column matrix with expandable/closeable items
- IP Address (Windows)

Comparing IUP with Other Interface Toolkits

Why to still maintain IUP if today we have so many other popular toolkits?

This is a question we always ask to ourselves before going on for another year.

To answer that question we must first define the characteristics of the reference toolkit, list the available toolkits and compare them with the reference and with IUP.

We would like a toolkit that has:

- **Portability.** That provides an abstraction for Graphical User Interface in Windows, UNIX and Macintosh. Also called Cross platform and multi-platform GUI toolkit.
- **Free License and Open Source.** This means that we can also produce commercial applications. The pure GPL license can not be used but the LGPL can but must contain an exception stating that derived works in binary form may be distributed on the user's own terms. This is a solution that satisfies those who wish to produce GPL'ed software and also those producing proprietary software. Many libraries are distributed with this license combination.
- **Small and Simple API.** This is rare. Many libraries assume that an Interface toolkit is also a synonymous of a system abstraction and accumulate thousands of extra functions that are not related to User Interface. At Tecgraf we like many small libraries instead of one big library. Almost all available toolkits today are in C++ only, so C applications are excluded, also this means a hundred classes to include and understand each member function. The use of attributes makes a lot of things more elegant and simpler to understand.
- **Native Look & Feel**. Many toolkits draw their own controls. This gives an uniformity among systems, but also a disparity among the available applications in the same system. Native controls are also faster because they are drawn by the system. But the problem is what's "native" in UNIX? Some commercial applications in UNIX start using Motif as the "native" option. It was the official standard but because of license restrictions, before the OpenMotif event, the system became old and some good alternatives were developed, including GTK and Qt.

Toolkits

With these characteristics in mind we select some of the available toolkits:

Name	License	Last Update	Version	Language	Platforms	Controls	Team	Comments
FOX	LGPL*	1997-2011/09	1.6.44	C++	Win, X	own	3	great look, license restrictions
FLTK	LGPL*	1998-2011/10	1.3.1	C++	Win, X, Mac	own	4	was from Digital Domain. Easy to learn.
GTK+	LGPL*	1997-2011/07	3.0.12	C	Win, X, Mac	own	12	target for X-Windows, basis of GNOME, Windows is apart, Mac using X
Qt	GPL	1994-2011/09	4.7.4	C++	Win, X, Mac	own	(many)	Is free for Non Commercial, a dual-licensing model, basis of KDE, Emulates the native look and feel
wxWidgets	LGPL*	1992-2011/07	2.9.2	C++	Win, X, Mac	native	6	X can use Motif or GTK, has many contributors
IUP	MIT	1994-2011/04	3.5	C	Win, X, Mac	native	2	X can use Motif or GTK, Mac using X

Table Last Update: November 2011

More toolkits can be found here: The GUI Toolkit, Framework Page and List of widget toolkits.
Interesting articles can be found here:

- GUI Toolkits for The X Window System
- Bad UI of the Week: The Cross-Platform User Interface
- Multi-platform User Interface Construction – a Challenge for Software Engineering-in-the-Small

Discussion

FOX has a great look but the license can be restrictive in some cases.

FLTK promises a new version with a better look and new features, but there are no concrete release dates. The FLTK documentation also does not help.

GTK+ can be used as a replacement for Motif, but not as a fully "portable" toolkit since it is was originally target for X-Windows. Nowadays GTK+ 2 is a great free C toolkit. But some predefined dialogs could be the native ones, like the File Selection, specially in Windows. The Windows port has a look and feel very similar to the Windows native look and feel, but it is different from a native application. A MacOS X port without using X-Windows is on the way, but very slowly. Unfortunately the Windows port has been orphan for some time and there is no release of new binaries for a while.

Qt had several license limitations, but since mid 2009 a new license model take place and it became more attractive. It is a very stable and powerful toolkit.

wxWidgets is an excellent choice because of the native controls and its portability.

It is hard to compare IUP with wxWidgets and Qt since they are much more than an User Interface Toolkit. They are complete development platforms that include several secondary libraries not related to User Interface. In IUP we focus only in Graphical User Interface.

Developing IUP

IUP has a C API, only has functions for Graphical User Interface, and uses "Native Controls" in Windows, Motif and GTK+. These are the major differences between IUP and other toolkits. Because of that IUP is small, fast and very powerful.

We have a small but very active team and we have many Tecgraf and foreign applications that today use IUP, collaborating for its evolution. Our objective is to surpass the Tecgraf needs, keeping backward compatibility and improving the internal code.

IUP does not have a wide localization feature, it only includes support for messages in English and Portuguese. And it does not have support for Unicode characters.

Why Not Mac? The first Mac driver was developed for MacOS 9 and had several memory limitations so it was abandoned. With Mac OS X we have the opportunity to do something better. Today IUP runs on Mac OS X using X11 with Motif or GTK. We plan for the future to build a native driver, but it is not a Tecgraf priority.

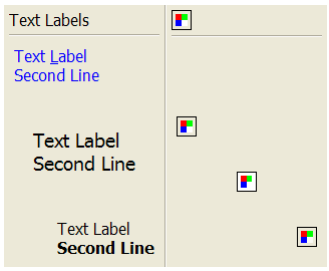
Why Still Motif? Motif is very important for non Linux systems, some Tecgraf applications run on old AIX, SGI and Sun systems, that only have Motif installed and we can not force the installation of other toolkits like GTK.

.. "Make it Reusable, Make it Simple, Make it Small" ...

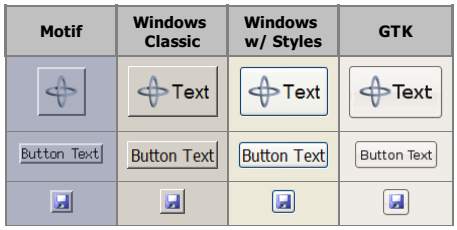
Gallery

Standard Controls

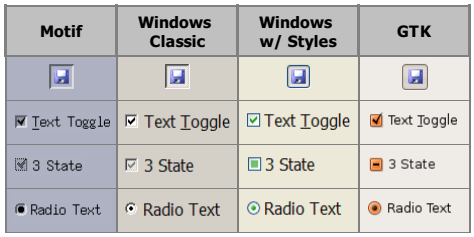
IupLabel



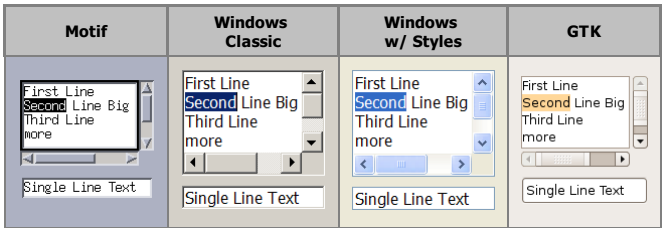
IupButton



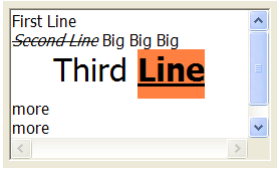
IupToggle



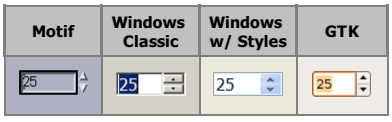
IupText



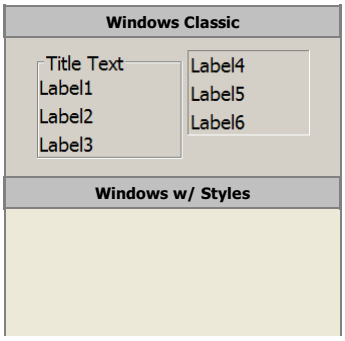
Using FORMATTING:

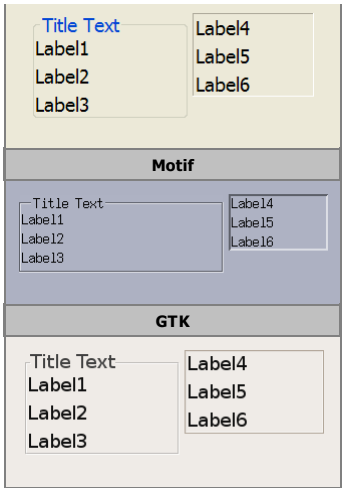


When SPIN=YES:

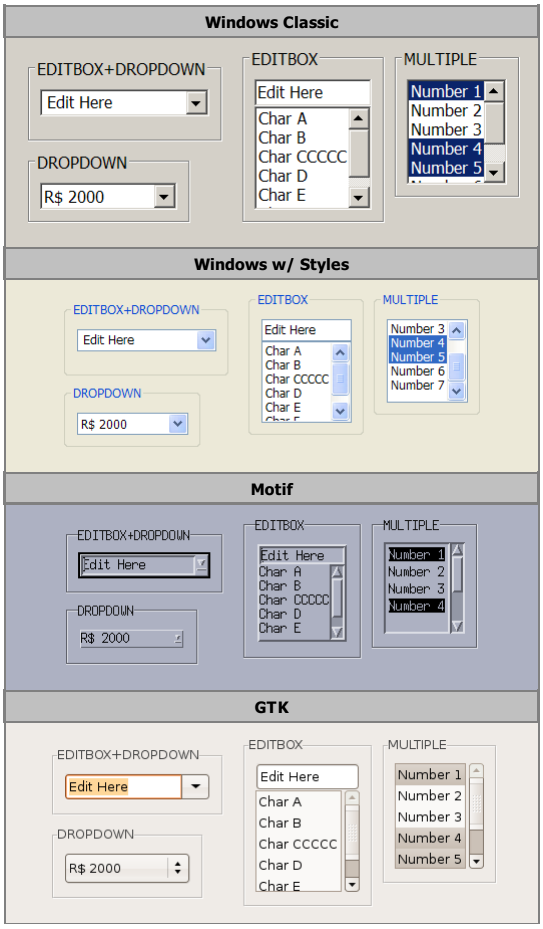


IupFrame

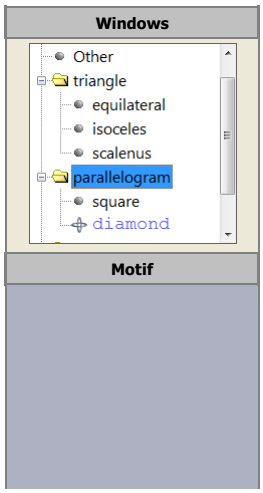


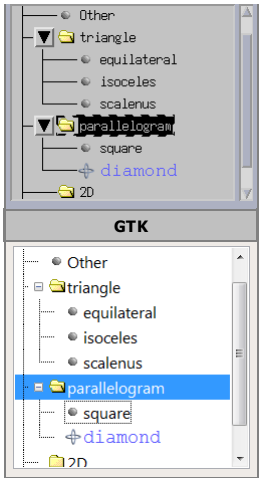


IupList

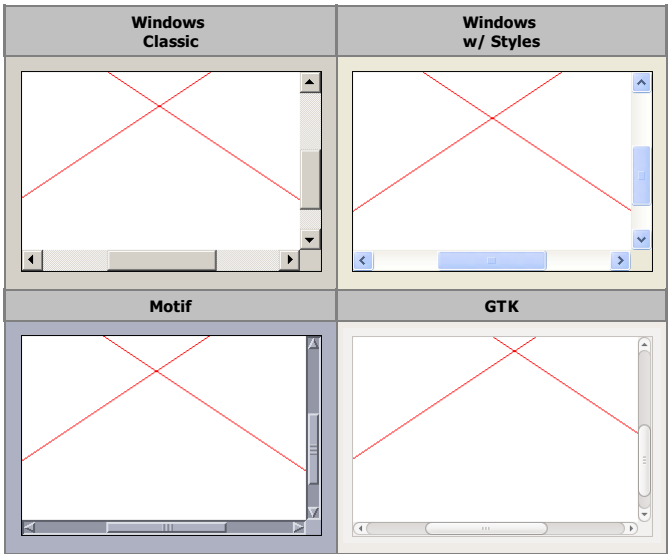


IupTree

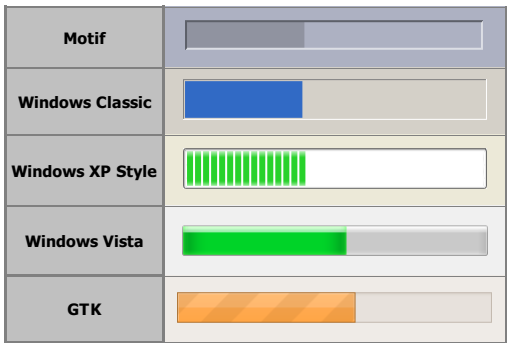




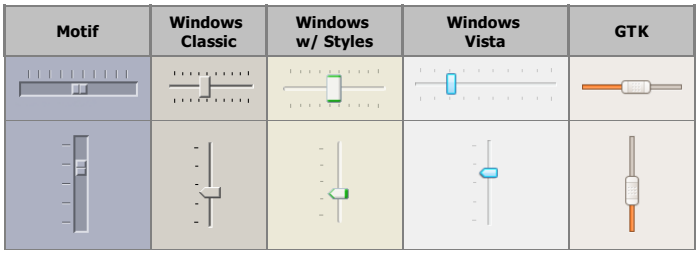
IupCanvas



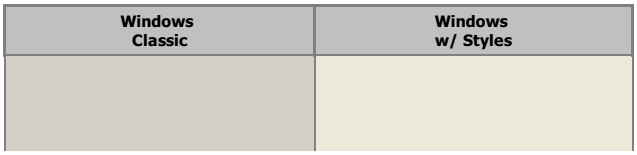
IupProgressBar

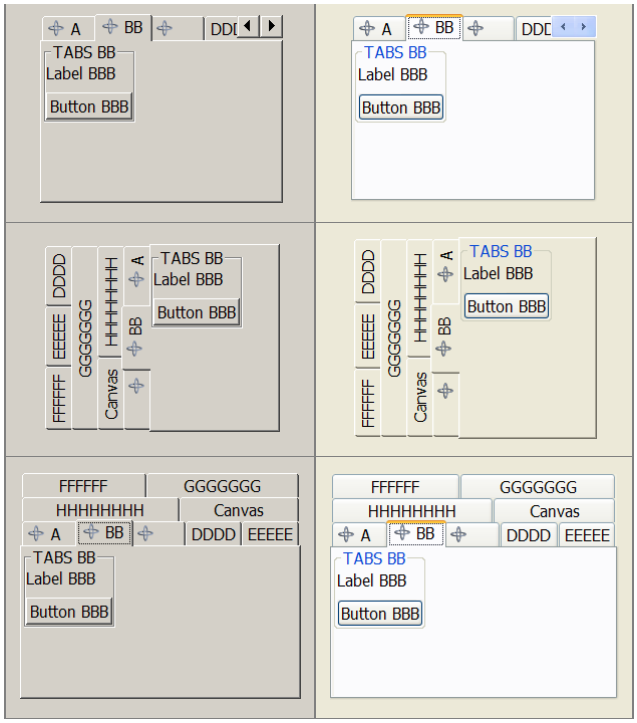


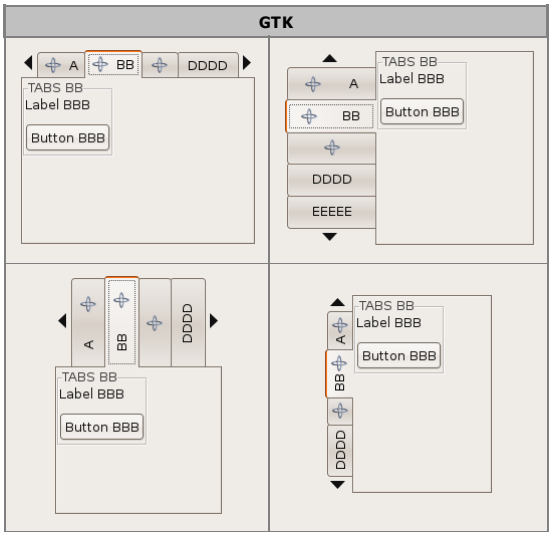
IupVal

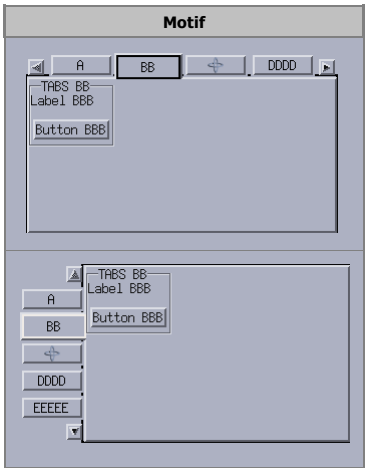


IupTabs

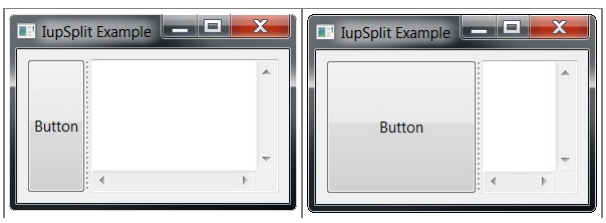



A
BB
DDD
TABS BB
Label BBB
Button BBB
DDDD
EEEEE
GGGGGGG
HHHHHHHH
Canvas
FFFFFF


GTK
A
BB
DDDD
EEEEE
TABS BB
Label BBB
Button BBB

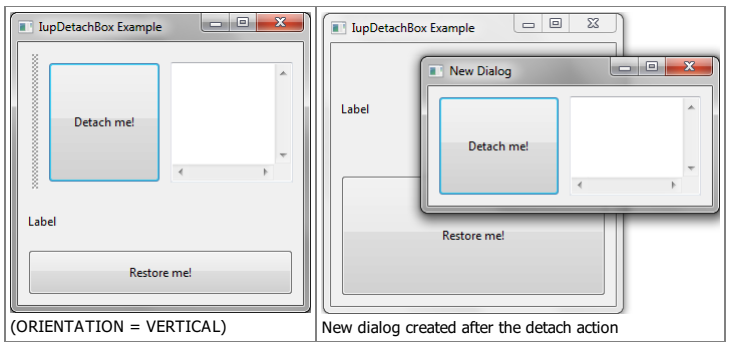

Motif
A
BB
DDDD
TABS BB
Label BBB
Button BBB
EEEEE

IupSplit


IupSplit Example
Button
IupSplit Example
Button

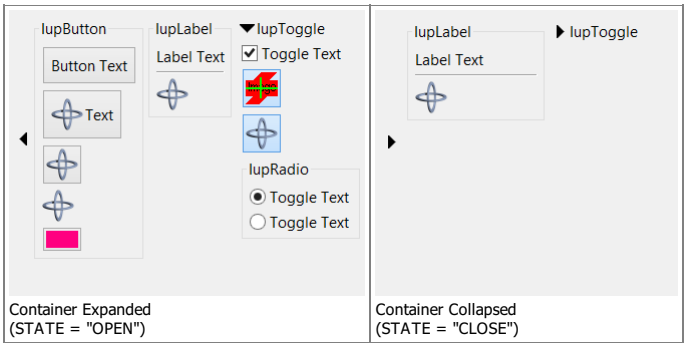
Natural Size	After Changing the Bar Position

IupDetachBox



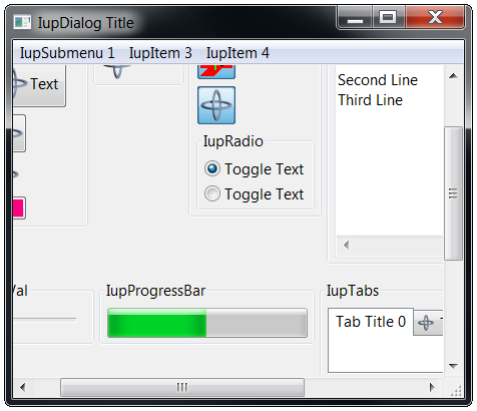
(ORIENTATION = VERTICAL) | New dialog created after the detach action

IupExpander



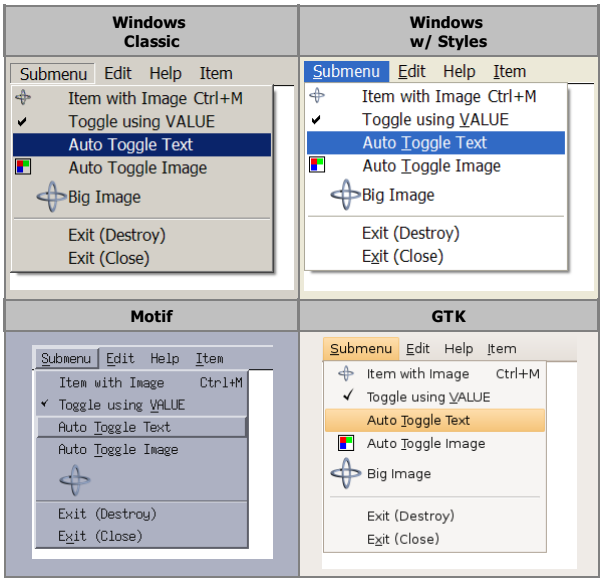
Container Expanded (STATE = "OPEN") | Container Collapsed (STATE = "CLOSE")

IupScrollBox

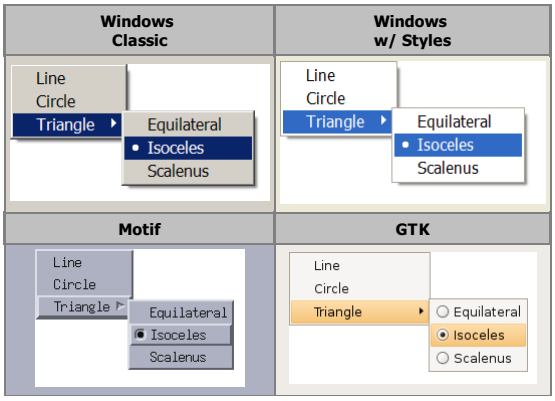


Resources

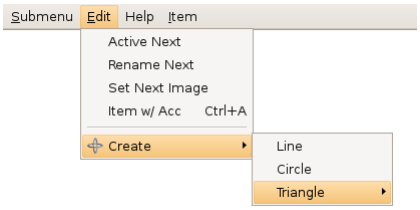
IupMenu, IupSubmenu and IupItem



The **IupItem** check is affected by the RADIO attribute in **IupMenu**:



Several Submenus:



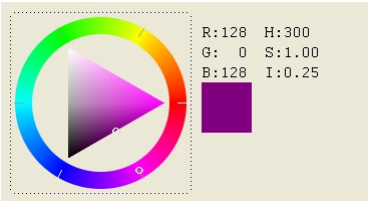
IupImage

See also the IupImgLib, a library of pre-defined images.

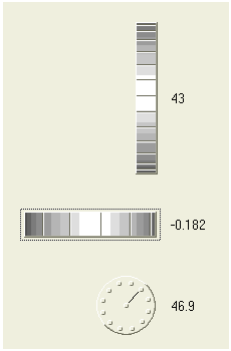
Gallery

Additional Controls

IupColorBrowser



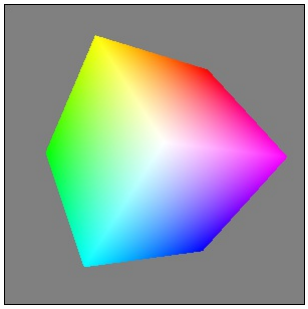
IupDial



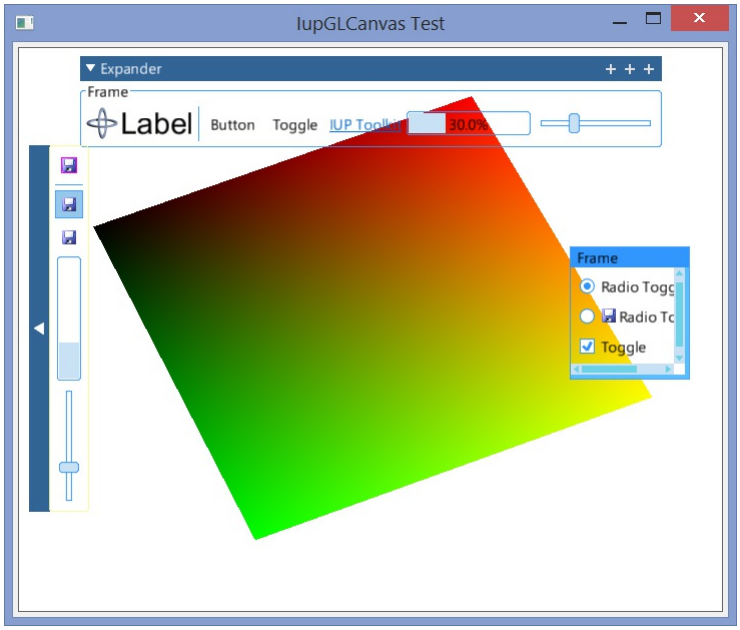
IupMatrix

Inflation	January 2000	February 2000
Medicine	5.6	4.5
Pharma	3.33	
Food	2.2	8.1
Energy	7.2	3.4

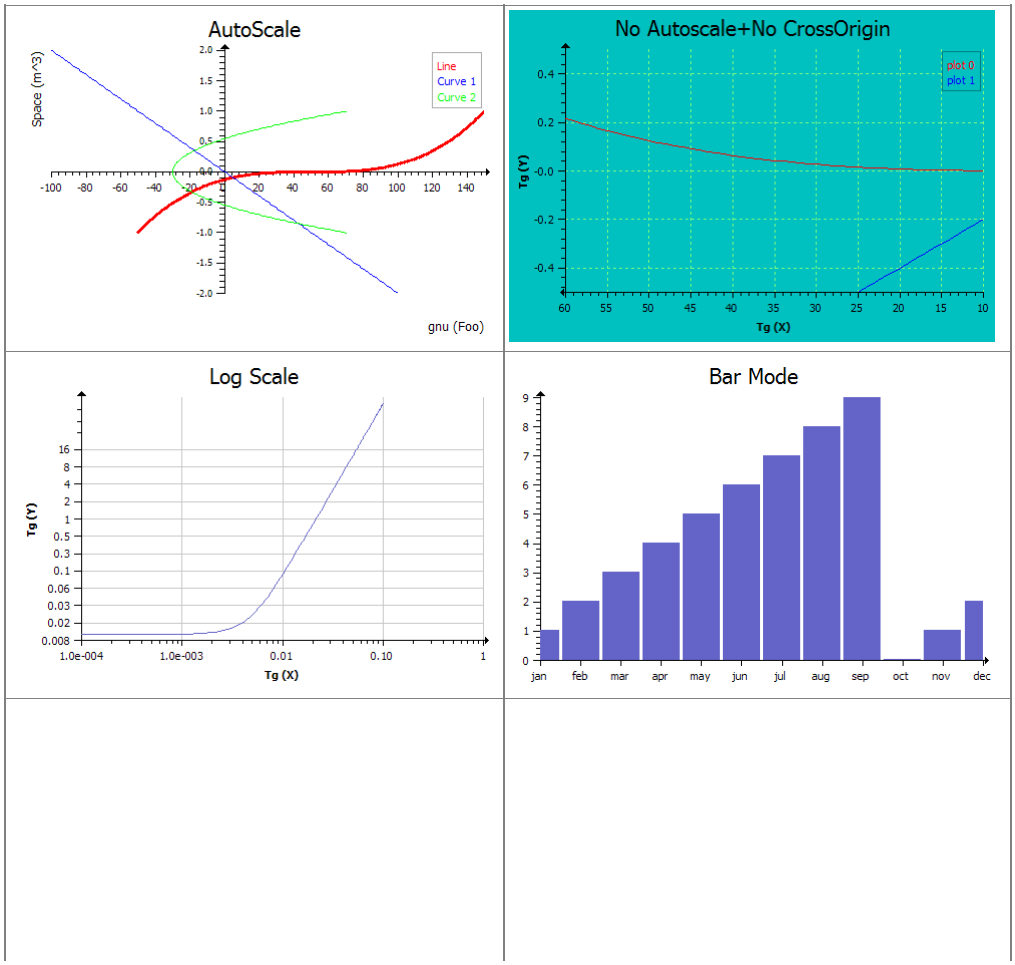
IupGLCanvas

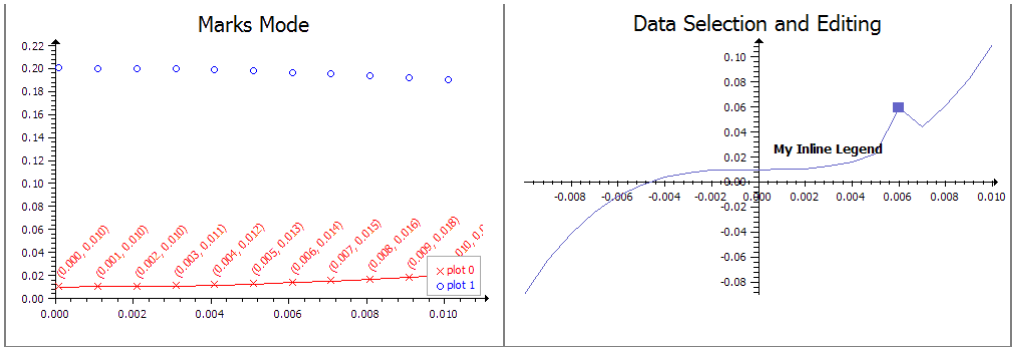


IupGLControls

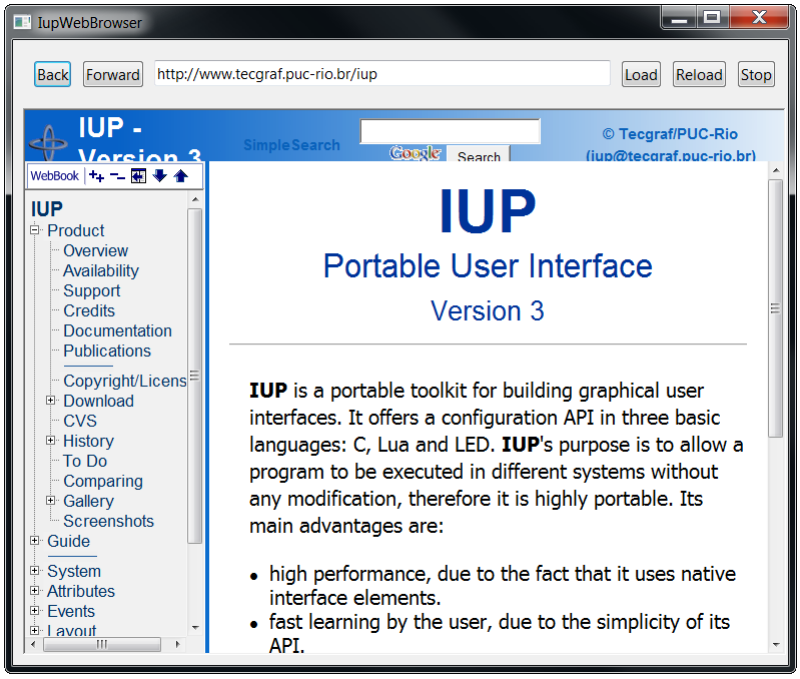


IupPPlot

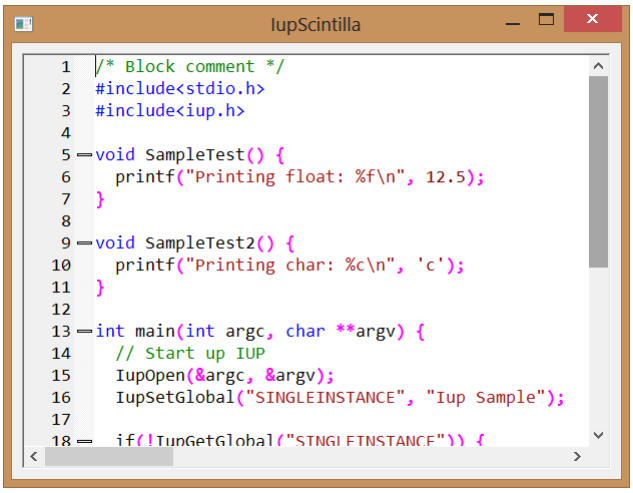




IupWebBrowser

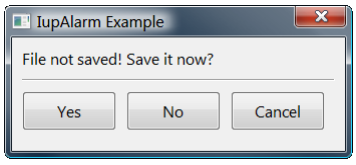


IupScintilla

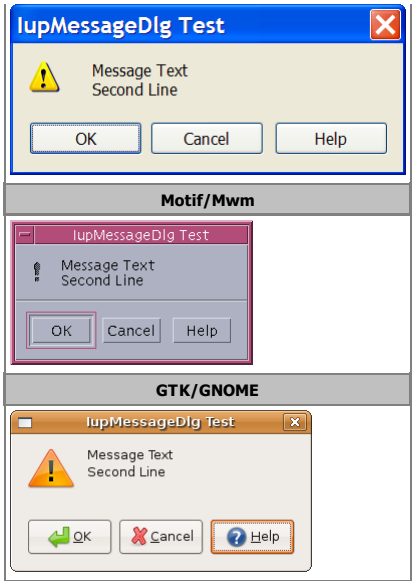


Pre-defined Dialogs

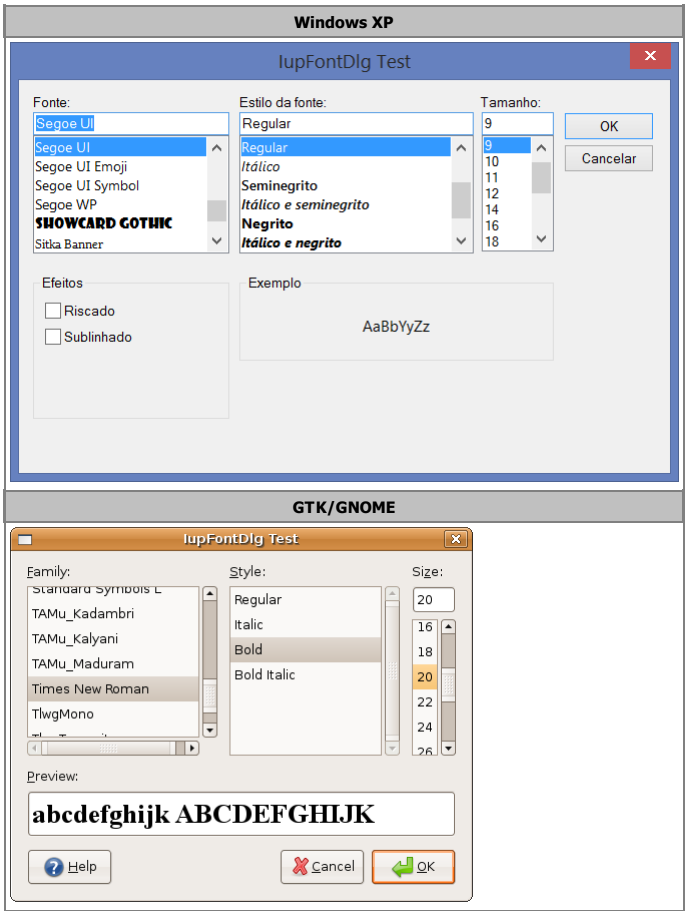
IupAlarm



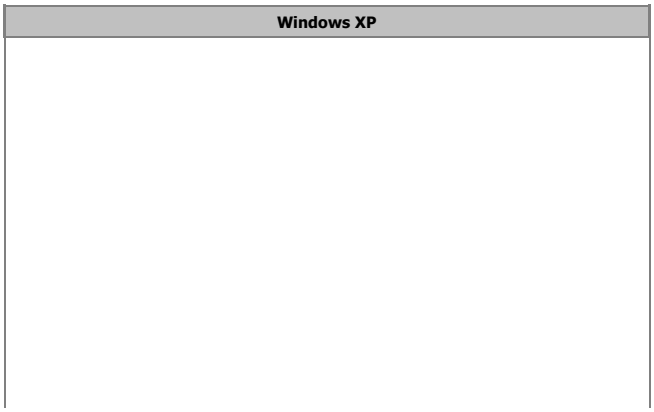
IupMessageDlg

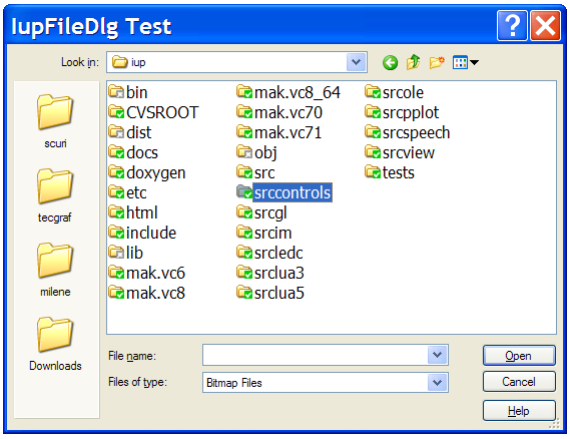


IupFontDlg

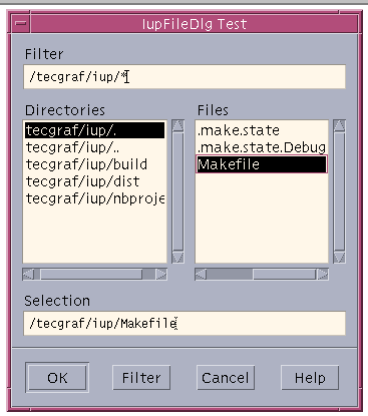


IupFileDlg

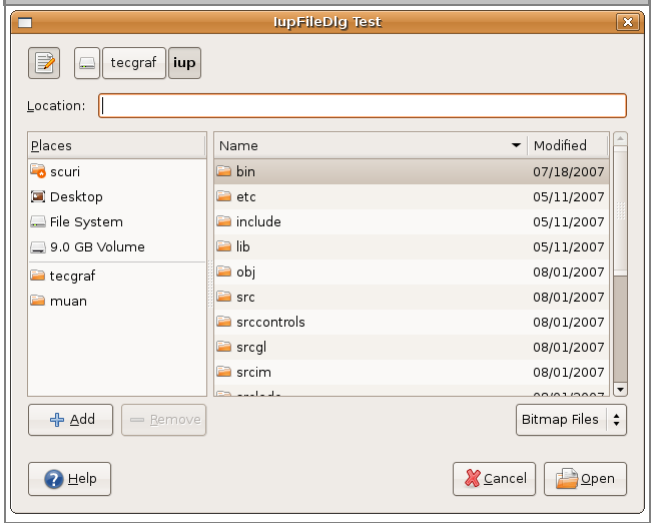




Motif/Mwm

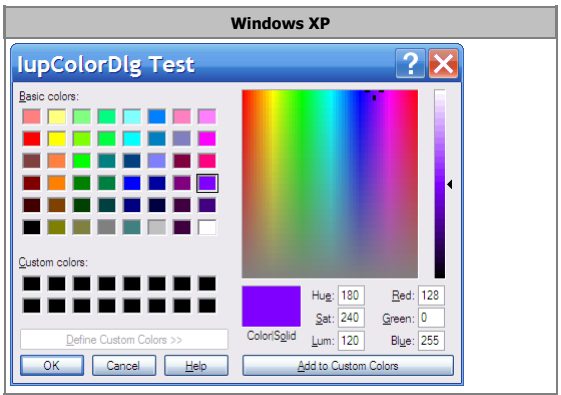


GTK/GNOME

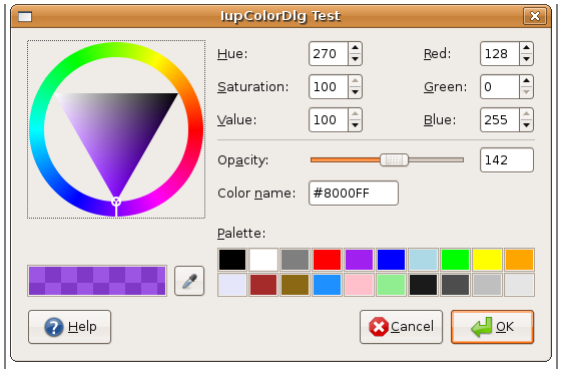


IupColorDlg

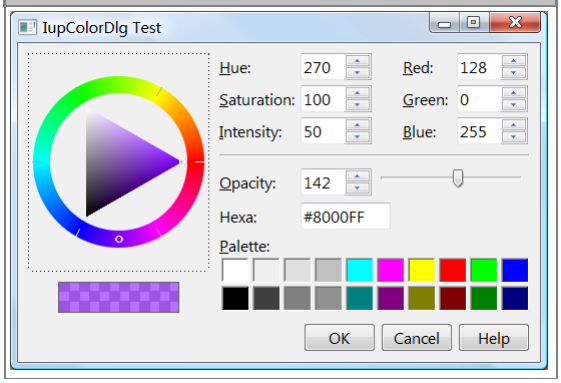
Windows XP



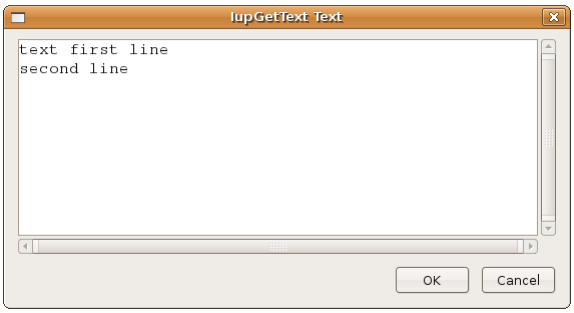
GTK/GNOME



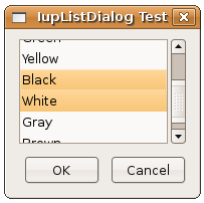
ColorBrowser Based



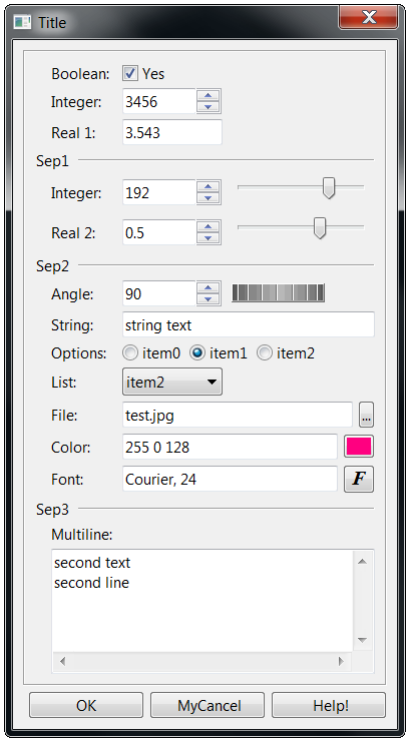
IupGetText



IupListDialog



IupGetParam

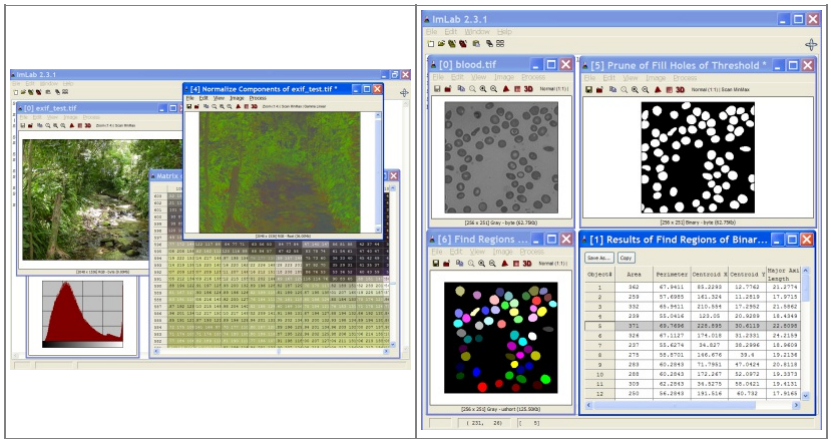


Screenshots

(Click on the picture to enlarge image.)

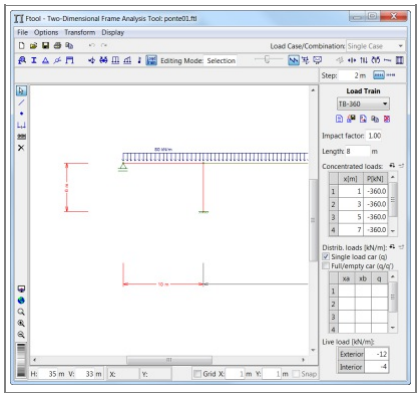
IMLAB

Image Processing Laboratory
http://imlab.sourceforge.net/



FTOOL

Two Dimensional Frame Analysis Tool
http://www.tecgraf.puc-rio.br/ftool

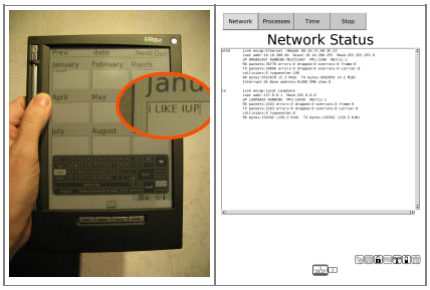


Nokia N800

IUP running on a Nokia N800 Internet Tablet using the GTK driver (contribution by Otfried Cheong).

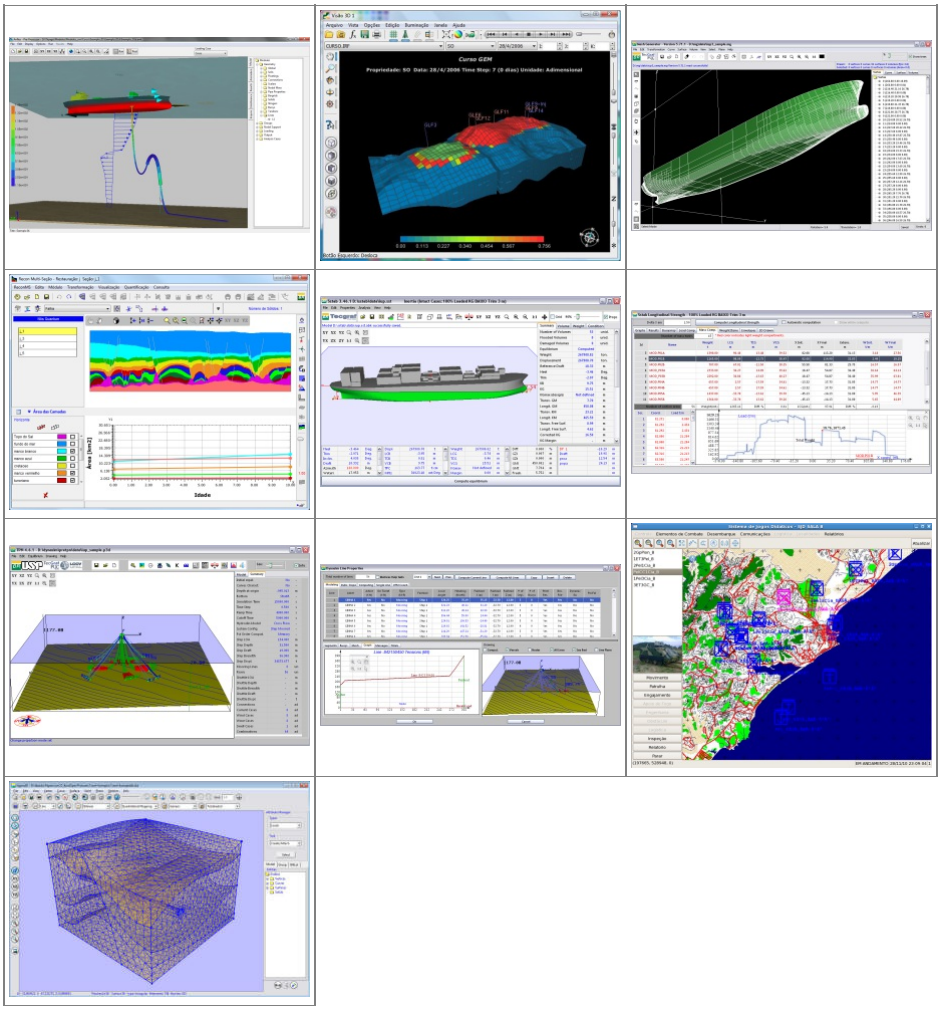
iRex Iliad Book Reader

IUP running on a iRex Iliad Book Reader using the GTK driver (contribution by Hans Elbers).

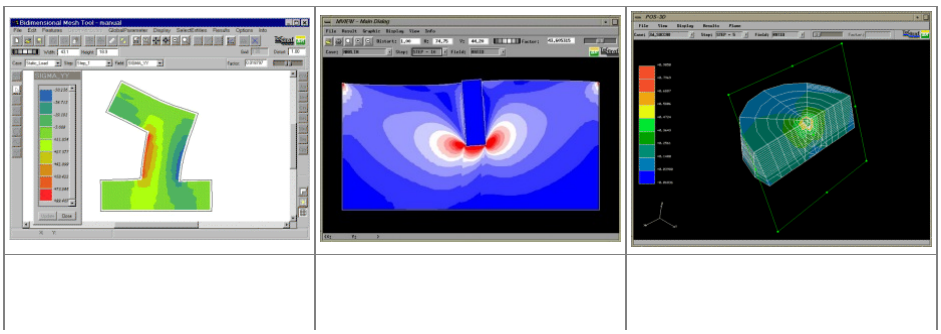


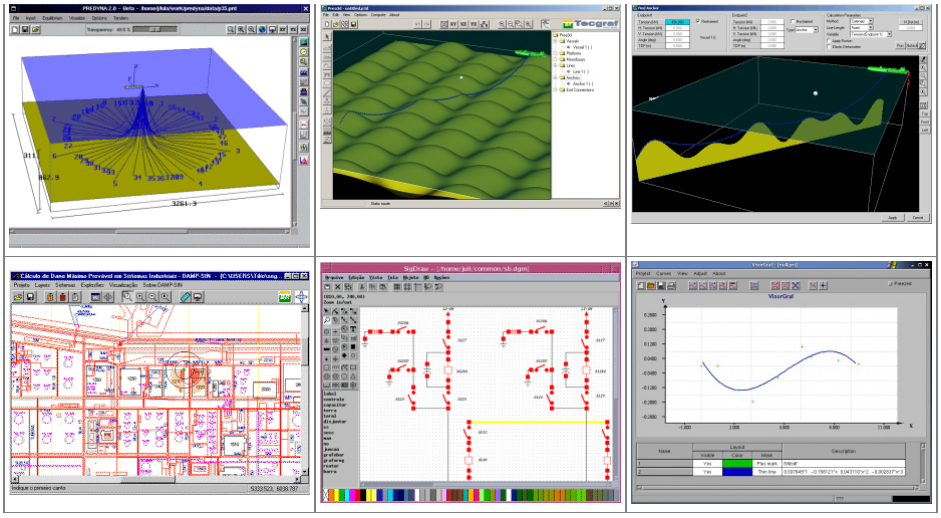
Tecgraf Applications

These are internal Tecgraf applications, developed across our partnerships. Their use is exclusive to the partner. The models seen in the screenshots are fake models used for this purpose alone.



Older Tecgraf Applications





Guide

Getting Started

IUP has four important concepts that are implemented in a very different way from other toolkits.

First is the control creation timeline. When a control is created it is not immediately mapped to the native system. So some attributes will not work until the control is mapped. The mapping is done when the dialog is shown or manually calling **IupMap** for the dialog. You can not map a control without inserting it into a dialog.

Second is the attribute system. IUP has only a few functions because it uses string attributes to access the properties of each control. So get used to **IupSetAttribute** and **IupGetAttribute**, because you are going to use them a lot.

Third is the abstract layout positioning. IUP controls are never positioned in a specific (x,y) coordinate inside the dialog. The positioning is always calculated dynamically from the abstract layout hierarchy. So get used to the **IupFill**, **IupHbox** and **IupVbox** controls that allows you to position the controls in the dialog.

Fourth is the callback system. Because of the LED resource files IUP has an indirect form to associate a callback to a control. You associate a C function with a name using **IupSetFunction**, and then associate the callback attribute with that name using **IupSetAttribute**. But applications now should use the **IupSetCallback** function to directly associate a callback for a control.

LED is the original IUP resource file which has been replaced in favor of Lua files, although it is still supported. But keep in mind that you can use IUP without using LED or Lua, by just using the C API.

Building Applications

To compile programs in C, simply include file **iup.h**. If the application only uses functions from IUP and other cross-platform libraries that have the same API in all platforms, then the application immediately becomes platform independent. The implementation of the IUP functions is different in each platform and the linker is in charge of solving the IUP functions using the library specified in the project/makefile. For further information on how to link your application, please refer to the specific driver documentation.

IUP can also work together with other interface toolkits. The main problem is the **IupMainLoop** function. If you are going to use only Popup dialogs, then it is very simple. But to use non modal dialogs without the **IupMainLoop** you must call **IupLoopStep** from inside your own message loop. Also it is not possible to use Iup controls with dialogs from other toolkits and vice-versa.

The generation of applications is highly dependent on each system, but at least the **iup** library must be linked.

To use the additional controls you will need the **iupcontrols** and **iupcd** libraries and the CD library **cd**.

Other controls are available in secondary libraries, they also may have other external dependencies, check the documentation of the control.

If you are using IUP libraries in Windows DLLs or in Posix SOs dynamic libraries, then it is not necessary to worry about secondary dependencies, only specify the libraries you directly use. If you link to the static libraries then you must include all the secondary dependencies.

To use the Lua Binding, you need to link the program with the **iuplua** library and with the **lua** library. The other secondary libraries also have their Lua binding libraries that must be linked to use the control in Lua.

The download files list includes the Tecgraf/PUC-Rio Library Download Tips document, with a description of all the available binaries.

Windows

For Windows, if you **statically** link the application with IUP you must link also with the libraries **ole32.lib** and **comctl32.lib** (provided with the compilers). The **iup.rc** resource file (or a custom version) should be included in the application's project/makefile so that some icons and cursors can be used when not using the DLLs and to enable Windows Visual Styles. **iup.rc** is located in "/etc" folder of the distribution.

There is also guides for using some IDEs: C++ Builder X, Dev-C++, OpenWatcom C++, Visual C++ 7 (Visual Studio 2003), Visual C++ 8 (Visual Studio 2005), Code Blocks and Eclipse for C++.

In Windows, when using Gcc to link an application the libraries order are as important as in UNIX. Always put the less dependent library at the end, for example:

```
-liup -lgdi32 -lcomdlg32 -lcomctl32 -luuid -loleaut32 -lole32
```

See more information in the Windows System Driver.

Motif

For Motif, IUP uses the Motif (Xm), the Xtoolkit (Xt) and the Xlib (X11) libraries. To **statically** link an application with IUP, use the following options in the linker call (in the same order):

```
-liup -lXm -lXmu -lXt -lX11 -lm
```

Though these are the minimum requirements, depending on the platform other libraries might be needed. Typically, they are X extensions (Xext), needed in SunOS, and Xpm (needed in Linux only). They must be listed after Xt and before X11. For instance:

```
-liup -lXm -lXpm -lXmu -lXt -lXext -lX11 -lm
```

Usually these libraries are placed in default directories, but you may require additional options:

Linux	-L/usr/X11R6/lib -I/usr/X11R6/include
IRIX	-L/usr/lib32 (X11) -L/usr/Motif-2.1/lib32 -I/usr/Motif-2.1/include (Motif)
SunOS	-L/usr/openwin/lib -I/usr/openwin/share/include (X11) -L/usr/dt/lib -I/usr/dt/share/include (Motif)

See more information in the Motif System Driver.

GTK+ (since IUP 3.0)

For GTK, IUP uses the GTK, GDK, Pango, Cairo if GTK 3, and GLib. To **statically** link an application with IUP, use the following options in the linker call (in the same order):

```
-lgtk-x11-2.0 -lgdk-x11-2.0 -lgdk_pixbuf-2.0 -lpangox-1.0 -lpango-1.0 -lgobject-2.0 -lgmodule-2.0 -lglib-2.0 -lXext -lX11 -lm  (for GTK 2)
or
-lgtk-3 gdk-3 -lgdk_pixbuf-2.0 -lpangocairo-1.0 -lpango-1.0 -lcairo   -lgobject-2.0 -lgmodule-2.0 -lglib-2.0 -lXext -lX11 -lm  (for GTK 3)
```

In UNIX the following INCLUDES paths are necessary:

```
/usr/include/atk-1.0 /usr/include/gtk-2.0 /usr/include/cairo /usr/include/pango-1.0 /usr/include/glib-2.0
and eventually:   /usr/lib/glib-2.0/include /usr/lib/gtk-2.0/include
            or   /usr/lib64/glib-2.0/include /usr/lib64/gtk-2.0/include
```

A simpler way to obtain GTK parameters for static linking is to use the **pkg-config** tool:

```
INCLUDES += $(shell pkg-config --cflags gtk+-2.0 gdk-2.0)
LIBS += $(shell pkg-config --libs gtk+-2.0 gdk-2.0)
```

See more information in the GTK System Driver.

Multithread

User interface is usually not thread safe and IUP is not thread safe. The general recommendation when you want more than one thread is to build the application and the user interface in the main thread, and create secondary threads that communicates with the main thread to update the interface. The secondary threads should not directly update the interface.

Dynamic Loading

Although we have dynamic libraries we do not recommend the dynamic loading of the main IUP library in Motif. This is because it depends on Motif and X11, you will have to load these libraries first. So it is easier to build a base application that already includes X11, Motif and the main IUP library than trying to load them all. In Windows this is not a problem.

The IUP secondary libraries can be easily dynamic loaded regardless of the system.

Building The Library

In the Downloads you will be able to find pre-compiled binaries for many platforms, all those binaries were built using Tecmake. Tecmake is a command line multi compiler build tool based on GNU make, available at http://www.tecgraf.puc-rio.br/tecmake. Tecmake is used by all the Tecgraf libraries and many applications.

You do not need to install Tecmake, scripts for Posix and Windows systems are already included in the source code package. Just type "make" in the command line on the main folder and all libraries and executables will be build.

In Linux, check the "Building Lua, IM, CD and IUP in Linux" guide.

In Windows, check the "Building Lua, IM, CD and IUP in Window" guide.

If you decide to install Tecmake, the Tecmake configuration files (*.mak) are available at the "src*" folders, and are very easy to understand. In the main folder, and in each source folder, there are files named *make_uname.bat* that build the libraries using **Tecmake**. To build for Windows using Visual C 9.0 (2008) for example, just execute *"make_uname vc9"* in the iup main folder, or for the DLLs type *"make_uname dll9"*. The Visual Studio workspaces with the respective projects available in the source package is for debugging purposes only.

IUP runs on many different systems and interact with many different libraries such as Motif, OpenGL, Canvas Draw (CD) and Lua. You have to install some these libraries to build the secondary IUP libraries. Make sure you have all the dependencies for the library you want installed, see the documentation below.

If you are going to build all the libraries, the makefiles and projects expect the following directory tree:

```
/mylibs/
        iup/
        cd/
        im/
        lua5.1/
```

To control that location set the TECTOOLS_HOME environment variable to the folder were the CD, IM and Lua libraries are installed.

IUP_ASSERT can be defined to enable some runtime checks for the main API.

Libraries Dependencies

```
iupwin* -> gdi32 user32 comdlg32 comctl32 uuid ole32 (system - Windows)
iupmot* -> [Xpm Xmu Xext] Xt X11 (system - UNIX)
iupgtk* -> gtk-win32-2.0 gdk-win32-2.0 pangowin32-1.0(system - Windows)
        -> gtk-x11-2.0 gdk-x11-2.0 pangox-1.0 (system - UNIX)
        -> gdk_pixbuf-2.0 pango-1.0 gobject-2.0 gmodule-2.0 glib-2.0 (system - Windows/UNIX)
iupgl -> iup
      -> opengl32 glu32 glaux (system - Windows)
      -> GLU GL (system - UNIX)
iupcd -> iup
      -> cd
iupcontrols -> iupcd
iup_pplot -> iupcd
          -> PPlot (included)
iupim -> iup
      -> im
iupimglib -> iup
iuplua51 -> iup
         -> lua5.1
iupluacd51 -> iuplua51
           -> cdlua51
           -> iupcd
iupluacontrols51 -> iuplua51
                 -> iupcontrols
iupluagl51 -> iuplua51
           -> iupgl
iupluaim51 -> iuplua51
           -> imlua51
```

```
          -> iupim
iupluaole51 -> iuplua51
          -> iupole
iuplua_pplot51 -> iuplua51
              -> iup_pplot
iupole -> iup
iupweb -> iupole (Windows)
       -> webkit-1.0 (Linux)

(*) In Windows, "iupwin" is called "iup".
    In Linux and BSD "iupgtk" is called "iup".
    In IRIX, AIX and SunOS "iupmot" is called "iup".
```

As a general rule (excluding system dependencies): IUP depends on CD and IM, and IUPLua depends on Lua, CDLua and IMLua. Notice that not all IUP libraries depend on CD and IM.

Instead of building all the libraries, try building only the libraries you are going to use. The Makefile on the root folder will build all the libraries, but in each source folder there are secondary Makefiles. We use the following source code structure:

```
iup/
    src/         - The core library. Motif, GTK and Windows code
    srccd/       - CD_IUP canvas driver for the CD library
    srcconsole/  - Lua interpreter executable with pre-loaded IUP, CD and IM libraries
    srcgl/       - IupGLCanvas
    srcim/       - IUP/IM utilities
    srcimglib/   - Image Libraries with Icons, Logos and Bitmaps
    srcledc/     - ledc executable
    srclua5/     - Lua 5 binding
    srcole/      - IupOleControl
    srcplot/     - IupPlot
    srctuio/     - IupTuioClient
    srcweb/      - IupWebBrowser
    srcview/     - IupView executable
```

The Lua bindings for IUP, CD and IM (Makefiles and Pre-compiled binaries) depend on the LuaBinaries distribution. So if you are going to build from source, then use the **LuaBinaries** source package also, not the **Lua.org** original source package. If you like to use another location for the Lua files define LUA_SUFFIX, LUA_INC, LUA_LIB and LUA_BIN before using Tecmake.

In Ubuntu you will need to install the following packages and their dependencies (they are not installed by default):

```
libgtk2.0-dev (for the GTK driver)
or
libgtk3.0-dev (for the GTK driver)
    libmotif-dev and x11proto-print-dev (for the Motif driver, if used)
libgl1-mesa-dev and libglu1-mesa-dev (for the IupGLCanvas)
libwebkitgtk-dev (for the IupWebBrowser)
or
libwebkitgtk3.0-dev (for the IupWebBrowser)
```

Using IUP in C++

IUP is a low level API, but at the same time a very simple and intuitive API. That's why it is implemented in C, to keep the API simple. But most of the actual IUP applications today use C++. To use C callbacks in C++ classes, you can declare the callbacks as static members or friend functions, and store the pointer "this" at the "Ihandle*" pointer as an user attribute. For example, you can create your dialog by inheriting from the following dialog.

```
class iupDialog
{
private:
  Ihandle *hDlg;
  int test;

  static int ResizeCB (Ihandle* self, int w, int h);
  friend int ShowCB(Ihandle *self, int mode);

public:
  iupDialog(Ihandle* child)
  {
    hDlg = IupDialog(child);
    IupSetAttribute(hDlg, "iupDialog", (char*)this);
    IupSetCallback(hDlg, "RESIZE_CB", (Icallback)ResizeCB);
    IupSetCallback(hDlg, "SHOW_CB", (Icallback)ShowCB);
  }

  void ShowXY(int x, int y) { IupShowXY(hDlg, x, y); }

protected:

  // implement this to use your own callbacks
  virtual void Show(int mode) {};
  virtual void Resize (int w, int h){};
};

int iupDialog::ResizeCB(Ihandle *self, int w, int h)
{
  iupDialog *d = (iupDialog*)IupGetAttribute(self, "iupDialog");
  d->test = 1; // private members can be accessed in private static members
  d->Resize(w, h);
  return IUP_DEFAULT;
}

int ShowCB(Ihandle *self, int mode)
{
  iupDialog *d = (iupDialog*)IupGetAttribute(self, "iupDialog");
  d->test = 1; // private members can be accessed in private friend functions
  d->Show(mode);
  return IUP_DEFAULT;
}
```

This is just one possibility on how to write a wrapper class around IUP functions. Some users contributed with C++ wrappers, see next on **Contributions**.

To help the creation of callbacks as methods in C++ Classes we provide (since 3.15) a header file called "iup_class_cbs.hpp" that has some macros for that usage. There are 3 macros that need to be called:

```
IUP_CLASS_INITCALLBACK - to register the object as a callback receiver (called only once)
IUP_CLASS_SETCALLBACK - to associate the callback for a given element
IUP_CLASS_DECLARECALLBACK_IFn* - to declare the callback as a member function
                                 (defined as the typedefs in "iupcbs.h" for all callbacks)
```

This allows an application to define callbacks as methods of any class it is using in the application. The method will have exactly the same definition as the callback. And it will not be a static method, so there will be no need to access a pointer to the this object. Here is an example:

```
class SampleClass
{
  int sample_count;

public:
  SampleClass()
  {
    sample_count = 0;

    Ihandle* button = IupButton("Test", NULL);
    // 2) Associate the callback with the button
    IUP_CLASS_SETCALLBACK(button, "ACTION", ButtonAction);

    Ihandle* dialog = IupDialog(button);
    // 1) Register this object as a callback receiver (only once)
    IUP_CLASS_INITCALLBACK(dialog, SampleClass);

    IupShow(dialog);
  };

protected:
  // 3) Declare the callback as a member function
  IUP_CLASS_DECLARECALLBACK_IFn(SampleClass, ButtonAction);
};

// 4) Define the callback as a member function
int SampleClass::ButtonAction(Ihandle*)
{
  sample_count++;

  IupExitLoop();
  return IUP_DEFAULT;
}
```

Contributions

All the contributions use the same license terms of the IUP license.

C++ Wrappers

RSSGui by Danny Reinhold. (RSS_GUI_0_5.zip)

Described by his words:
- It works fine with the C++ STL and doesn't define a set of own string, list, vector etc. classes like many other toolkits do (for example wxWidgets).
- It has a really simple event handling mechanism that is much simpler than the system that is used in MFC or in wxWidgets and that doesn't require a preprocessor like Qt. (It could be done type safe using templates as in a signal and slot library but the current way is really, really simple to understand and to write.)
- It has a Widget type for creating wizards.
- It is not complete, some things are missing. It was tested only on the Windows platform.

IupTreeUtil by Sergio Maffra and Frederico Abraham. (IupTreeUtil3.zip or IupTreeUtil3.tar.gz)

It is an utility wrapper for the **IupTree** control. It has several limitations, including to add leaves only after all branches inside a branch. It uses STL.

IUP with C++ 11 and variadic templates (IUP++) by PulkoMandy

The IUP++ class registers itself as an IUP callback (with any arguments) and forwards the call to a C++ object and method.

Tools

IupAsync by Ross Berteig

Described by his words:
IUP is not designed to be accessed from multiple threads, but occasionally there is a need (especially in a multi-threaded application) for the UI to update a display or dispatch an action in response to messages from other threads or from an OS component. To address this need, we designed an IUP control that translates calls from any application thread into a callback function guaranteed to be running in IUP's thread.
The IupAsync control is presently an alpha release proving the concept for the Windows platform only. It is intended that it be ported to the other platforms supported by IUP (GTK and Motif for Linux and OSX).

Drivers

IUP 3 MacOSX Driver by Heesob Park

A native driver for MacOSX using Cocoa. On going work. Help needed!

Language Bindings

A Basic Guide to using IupLua by Steve Donovan

A very nice introductory tutorial for IupLua.

Ruby-IUP by Heesob Park

ruby-iup is an extension module for Ruby that provides an interface to the IUP GUI toolkit. The source is hosted on github.com at http://github.com/phasis68/ruby-iup.

EuIup by Jeremy Cowgar

IUP wrapped for Euphoria.

FreeBasic Binding by AGS

The first release of FreeBASIC bindings for IUP 3. See the Forum post Portable GUI toolkit (IUP) version 3.0 (RC2)

Perl Binding by Kmx

Perl binding for IUP and related libraries.

Go-iup by Jeremy Cowgar

IUP wrapped for Go.

ScriptBasic Binding by John Spikowski

ScriptBasic binding for IUP. See the Forum posts about the Extension Module at IUP.

Component Pascal Binding by Boris Ilov

Component Pascal binding for IUP and CD, part of the BlackBox Framework.

Building Lua, IM, CD and IUP in Linux

This is a guide to build all the Lua, IM, CD and IUP libraries in Linux. Notice that you may not use all the libraries, although this guide will show you how to build all of them. You may then choose to build specific libraries.

The Linux used as reference is the Ubuntu distribution.

System Configuration

To build the libraries you will have to download the development version of some packages installed on your system. Although the run time version of some of these packages are already installed, the development versions are usually not. The packages described here are for Ubuntu, but you will be able to identify them for other systems as well.

To build Lua you will need:

```
libreadline-dev
```

To build IM you will need:

```
g++
```

To build CD you will need:

```
libfreetype6-dev
libgl1-mesa-dev and libglu1-mesa-dev (for the ftgl library used by CD_GL)
libgtk2.0-dev (for the GDK driver)
or
libgtk3.0-dev (for the GTK driver)
  libx11-dev (for the X11 driver, OPTIONAL)
  libxpm-dev (")
  libxmu-dev (")
  libxft-dev (for the XRender driver, OPTIONAL)
```

To build IUP you will need:

```
libgtk2.0-dev (for the GTK driver)
or
libgtk3.0-dev (for the GTK driver)
  libmotif-dev and x11proto-print-dev (for the Motif driver, OPTIONAL)
libgl1-mesa-dev and libglu1-mesa-dev (for the IupGLCanvas)
libwebkitgtk-dev (for the IupWebBrowser)
or
libwebkitgtk3.0-dev (for the IupWebBrowser)
```

To install them you can use the Synaptic Package Manager and select the packages, or can use the command line and type:

```
sudo apt-get install package_name
```

Source Download

Download the "xxx-X.X_Sources.tar.gz" package from the "**Docs and Sources**" directory for the version you want to build. Here are links for the **Files** section in **Source Forge**:

Lua - http://sourceforge.net/projects/luabinaries/files/
IM - http://sourceforge.net/projects/imtoolkit/files/
CD - http://sourceforge.net/projects/canvasdraw/files/
IUP - http://sourceforge.net/projects/iup/files/

Unpacking

To extract the files use the tar command at a common directory, for example:

```
mkdir -p xxxx
cd xxxx

[copy the downloaded files, to the xxxx directory]

tar -xpvzf lua5_1_4_Sources.tar.gz    [optional, see note below]
tar -xpvzf im-3.6.2_Sources.tar.gz
tar -xpvzf cd-5.4_Sources.tar.gz
tar -xpvzf iup-3.2_Sources.tar.gz
```

If you are going to build all the libraries, the makefiles and projects expect the following directory tree:

```
/xxxx/
      lua5.1/    [optional, see note below]
      im/
      cd/
      iup/
```

If you unpack all the source packages in the same directory, that structure will be automatically created.

If you want to use some of these libraries that are installed on the system (see Installation section below) you will have to define some environment variables before building them. For example:

```
export IM_INC=/usr/include/im
export IM_LIB=/usr/lib           [not necessary, already included by gcc]

export CD_INC=/usr/include/cd
export CD_LIB=/usr/lib           [not necessary, already included by gcc]

export IUP_INC=/usr/include/iup
export IUP_LIB=/usr/lib          [not necessary, already included by gcc]
```

Lua

Although we use Lua from LuaBinaries, any Lua installation can also be used. In Ubuntu, the Lua run time package is:

```
lua5.1
```

And the Lua development package is:

```
liblua5.1-0-dev
```

To use them, instead of using the directory "/xxxx/lua5.1" described above, you will have to define some environment variables before building IM, CD and IUP:

```
export LUA_SUFFIX=
export LUA_INC=/usr/include/lua5.1
```

By default the Makefiles and Tecmake files will build for Lua 5.1. To build for other Lua versions define USE_LUA52=Yes or USE_LUA53=Yes in the environment.

Building

As a general rule (excluding system dependencies): IUP depends on CD and IM, and CD depends on IM. So start by build IM, then CD, then IUP.

To start building go the the "**src**" directory and type "**make**". In IUP there are many "srcxxx" folders, so go to the up directory "iup" and type "**make**" that all the sub folders will be built. For example:

```
cd im/src
make
cd ../..

cd cd/src
make
cd ../..

cd iup
make
cd ..
```

TIP: Instead of building all the libraries, try building only the libraries you are going to use. The provided makefiles will build all the libraries, but take a look inside them and you will figure out how to build just one library.

TIP: If GTK headers or libraries are not being found, even when the libgtk2.0-dev package is installed, then their installation folder is not where our Makefiles expect. Build the GTK/GDK dependent libraries using "make USE_PKGCONFIG=Yes".

Pre-compiled Binaries

Instead of building from sources you can try to use the pre-compiled binaries. Usually they were build in the latest Ubuntu versions for 32 and 64 bits. The packages are located in the "**Linux Libraries**" directory under the **Files** section in **Source Forge**, with **"xxx-X.X_Linux26g4_lib.tar.gz"** and **"xxx-X.X_Linux26g4_64_lib.tar.gz"** names.

Do not extract different pre-compiled binaries in the same directory, create a subdirectory for each one, for example:

```
mkdir im
cd im
tar -xpvzf ../im-3.6.2_Linux26g4_lib.tar.gz
cd ..

mkdir cd
cd cd
tar -xpvzf ../cd-5.4_Linux26g4_lib.tar.gz
cd ..

mkdir iup
cd iup
tar -xpvzf ../iup-3.2_Linux26g4_lib.tar.gz
cd ..
```

For the installation instructions below, remove the "lib/Linux26g4" from the following examples if you are using the pre-compiled binaries.

Installation (System Directory)

After building you can copy the libraries files to the system directory. If you are inside the main directory, to install the dynamic libraries you can type, for example:

```
sudo cp -f im/lib/Linux26g4/*.so /usr/lib              [script version: install ]
sudo cp -f cd/lib/Linux26g4/*.so /usr/lib
sudo cp -f iup/lib/Linux26g4/*.so /usr/lib
```

To install the development files, then do:

```
sudo mkdir -p /usr/include/im                          [script version: install_dev ]
sudo cp -fR im/include/*.h /usr/include/im
sudo cp -f im/lib/Linux26g4/*.a /usr/lib

sudo mkdir -p /usr/include/cd
sudo cp -f cd/include/*.h /usr/include/cd
sudo cp -f cd/lib/Linux26g4/*.a /usr/lib

sudo mkdir -p /usr/include/iup
sudo cp -f iup/include/*.h /usr/include/iup
sudo cp -f iup/lib/Linux26g4/*.a /usr/lib
```

Then in your makefile use -Iim -Icd -Iiup for includes. There is no need to specify the libraries directory with -L. Development files are only necessary if you are going to compile an application or library in C/C++ that uses there libraries. To just run Lua scripts they are not necessary.

Installation (Build Directory) [Alternative]

If you **don't** want to copy the dynamic libraries to your system directory, you can use them from build directory. You will need to add the dynamic libraries folders to the LD_LIBRARY_PATH (DYLD_LIBRARY_PATH in MacOSX), for example:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/xxxx/im/lib/Linux26g4:/xxxx/cd/lib/Linux26g4:/xxxx/iup/lib/Linux26g4
or for the current folder
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

And in your makefile will will also need to specify those paths when linking using -L/xxxx/iup/lib/Linux26g4, and for compiling use -I/xxxx/iup/include.

Installation (Lua Modules)

Lua modules in Ubuntu are installed in the "/usr/lib/lua/5.1" directory. So to be able to use the Lua "require" with IUP, CD and IM you must create symbolic links inside that directory.

```
sudo mkdir -p /usr/lib/lua/5.1                        [script version: config_lua_module ]
cd /usr/lib/lua/5.1

sudo ln -fs /usr/lib/libiuplua51.so iuplua.so
sudo ln -fs /usr/lib/libiupluacontrols51.so iupluacontrols.so
...
```

Using those links you do not need any extra configuration.

Installation (Lua Modules) [Alternative]

If you use the **alternative** installation directory, and you also do NOT use the LuaBinaries installation, then you must set the LUA_CPATH environment variable:

```
export LUA_CPATH=./\?.so\;./lib\?.so\;./lib\?51.so\;
```

Building Lua, IM, CD and IUP in Windows

This is a guide to build all the Lua, IM, CD and IUP libraries in Windows. Notice that you may not use all the libraries, although this guide will show you how to build all of them. You may then choose to build specific libraries.

System Configuration

The Tecmake configuration files are for the GNU **make** tool. So first the GNU **make** must be installed, and it must be in the PATH before other makes. MingW, Cygwin and GnuWin32 distributions have GNU **make** binaries ready for download. Some utilities are also necessary, specially to build the dependencies file: **mkdir**, **rm** (both in CoreUtils) **[**, **which, sed** and **g++]**. If you don't need the dependencies or some other options just ignore them. Also some features will work best if **bash** is installed.

When installing **Cygwin** unmark all pre-selected items. This is easier to do in "Partial" mode view. Then select only "**make**", it will automatically select other packages that "**make**" depends on. And select the **mkdir**, **rm**, **which, sed** and **g++** packages. Change PATH in "Control Panel/System/Advanced/Environment Variables" and add "c:\cygwin\bin;".

When installing **MingW** mark C Compiler, C++ Compiler, MSYS Basic System, and MinGW Developer Toolkit. Change PATH in "Control Panel/System/Advanced/Environment Variables" and add "C:\mingw4\msys\1.0\bin;C:\mingw4\bin;".

For **GnuWin32** it is faster to manually download and install the selected tools packages. But it does not include a compiler and does not include bash.

Win-Bash contains a "Shell-Complete" distribution and can also be used. It contains all the tools and bash. It does not include a compiler.

Finally install the compiler of your choice, among the following supported compilers:

- Visual C++ or just the Windows SDK.
- Gnu gcc (MingW or Cygwin)
- Open Watcom C++
- Embarcadero C++ (ex-Borland)

Tecmake Configuration

Since the compilers in Windows are not in the path, you must set a few environment variables to configure their location. For example:

```
VC10=c:/progra~2/micros~1/vc              (C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC)
VC10SDK=c:/progra~1/micros~1/Windows\v7.1  (C:\Program Files\Microsoft SDKs\Windows\v7.1)
                                 (if you only installed the Windows SDK with its own compiler set,
                                  then set both variables to the same location)
                                 (VC9,VC9SDK,VC8 and PLATSDK can also be set)
MINGW4=c:/mingw
GCC4=c:/cygwin
OWC1=d:/lng/owc1
BC6=d:/lng/bc6
```

Noticed that the path used can not have spaces because of the GNU make internal processing. So you should install or copy the compiler files to a path with no spaces, or you can use the short path name instead as in the example above. To obtain the short path name you can use the "shortpath.exe" Tecmake utility or use the CMD command “dir /X”.

If you installed the Visual Studio compiler set, then to use it in the command line run the "Visual Studio Command Prompt" item in the "Microsoft Visual Studio 2010\Visual Studio Tools" start menu.

In Windows, there are several compilers that build for the same platform. So when using the Makefiles included in the distributions of those libraries you must first specify which compiler you want to use. To do that set the TEC_UNAME environment variable. This variable will also define if you are going to build static or dynamic (DLL) libraries, and if building 32 or 64 bits binaries. For example:

```
TEC_UNAME=vc10      (Visual C++ 10, static library, 32bits)
TEC_UNAME=dll10     (Visual C++ 10, dynamic library, 32bits)
TEC_UNAME=vc10_64   (Visual C++ 10, static library, 64bits)
TEC_UNAME=dll10_64  (Visual C++ 10, dynamic library, 64bits)
TEC_UNAME=mingw4    (MingW gcc 4, static library, 32bits)
TEC_UNAME=dllw4     (MingW gcc 4, dynamic library, 32bits)
TEC_UNAME=gcc4      (Cygwin Win32 gcc 4, static library, 32bits)
TEC_UNAME=cygw17    (Cygwin Posix gcc 4, both static and dynamic libraries, 32bits)
TEC_UNAME=owc1      (Open Watcom C++ 1, static library, 32bits)
TEC_UNAME=bc6       (Embarcadero C++ 6, static library, 32bits)
```

Here is an example for MingW:

```
Download MingW installation tool:
  http://sourceforge.net/projects/mingw/files/Installer/mingw-get-inst/
Install MingW:
  Select C and C++ Compiles, MSYS Basic System, and MinGW Developer Toolkit.
Configure Environment (Minimum):
  set PATH=C:\mingw4\msys\1.0\bin;C:\mingw4\bin;%PATH%
  set MINGW4=c:/mingw4
  set TEC_UNAME=mingw4
Start Building:
  make
```

Here is an example for Visual C++:

```
Download Visual C++ Express edition:
```

```
  http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-for-windows-desktop
Install Visual C++ and the Windows SDK (also called Platform SDK)
Download Cygwin:
  http://www.cygwin.com/
Install Cygwin:
  Unselect all option, and select only "make"
Configure Environment (Minimum):
  set PATH=C:\cygwin\bin;%PATH%
  set VC9=C:/PROGRA~1/MICROS~1.0/VC
  set VC9SDK=C:/PROGRA~1/MICROS~2/Windows/v6.0A
Run the "CMD Shell" or "Build Environment" item in the Start Menu.
  or manually run the vcvars32.bat or vcvars64.bat script
  just once, before building any of the targets.
```

If not using the vcvars*.bat configuration scripts, then you must also set PATH:

```
REM The first two are just auxiliary variables.
set VS9=C:\Program Files (x86)\Microsoft Visual Studio 9.0
set VS9SDK=C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin
set PATH=%VS9%\Common7\IDE;%VS9%\VC\BIN;%VS9%\Common7\Tools;%VS9%\Common7\Tools\bin;%VS9%\VC\VCPackages;%VS9SDK%\bin;%PATH%
```

Source Download

Download the "xxx-X.X_Sources.tar.gz" package from the "**Docs and Sources**" directory for the version you want to build. Here are links for the **Files** section in **Source Forge**:

Lua - http://sourceforge.net/projects/luabinaries/files/
IM - http://sourceforge.net/projects/imtoolkit/files/
CD - http://sourceforge.net/projects/canvasdraw/files/
IUP - http://sourceforge.net/projects/iup/files/

Unpacking

To extract the files use the tar command at a common directory, for example:

```
mkdir -p xxxx
cd xxxx

[copy the downloaded files, to the xxxx directory]

tar -xpvzf lua5_1_4_Sources.tar.gz    [optional, see note below]
tar -xpvzf im-3.6.2_Sources.tar.gz
tar -xpvzf cd-5.4_Sources.tar.gz
tar -xpvzf iup-3.2_Sources.tar.gz
```

If you are going to build all the libraries, the makefiles and projects expect the following directory tree:

```
/xxxx/
      lua5.1/
      im/
      cd/
      iup/
```

If you unpack all the source packages in the same directory, that structure will be automatically created.

By default the Makfiles and Tecmake files will build for Lua 5.1. To build for other Lua versions define USE_LUA52=Yes or USE_LUA53=Yes in the environment.

Building

As a general rule (excluding system dependencies): IUP depends on CD and IM, and CD depends on IM. So start by build IM, then CD, then IUP.

To start building go the the "**src**" directory and type "**make**". In IUP there are many "srcxxx" folders, so go to the up directory "iup" and type "**make**" that all the sub folders will be built. For example:

```
cd im/src
make
cd ../..

cd cd/src
make
cd ../..

cd iup
make
cd ..
```

TIP: Instead of building all the libraries, try building only the libraries you are going to use. The provided makefiles will build all the libraries, but take a look inside them and you will figure out how to build just one library.

Pre-compiled Binaries

Instead of building from sources you can try to use the pre-compiled binaries. Usually they were build in the latest Windows versions for 32 and 64 bits. The packages are located in the "**Windows Libraries**" directory under the **Files** section in **Source Forge**, with **"xxx-X.X_Win32_xx_lib.tar.gz"** and **"xxx-X.X_Win64_xx_lib.tar.gz"** names.

Do not extract different pre-compiled binaries in the same directory, create a subdirectory for each one, for example:

```
mkdir im
cd im
tar -xpvzf ../im-3.6.2_Win32_vc10_lib.tar.gz
cd ..

mkdir cd
cd cd
tar -xpvzf ../cd-5.4_Win32_vc10_lib.tar.gz
cd ..

mkdir iup
cd iup
tar -xpvzf ../iup-3.2_Win32_vc10_lib.tar.gz
cd ..
```

Usage

For makefiles use:

```
1) "-I/xxxx/iup/include" to find include files
2) "-L/xxxx/iup/lib/vc10" to find library files
3) "-liup" to specify the library files
```

For IDEs the configuration involves the same 3 steps above, but each IDE has a different dialog. The IUP toolkit has a Guide for some IDEs:

Borland C++ BuilderX - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/cppbx.html
Code Blocks - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/codeblocks.html
Dev-C++ - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/dev-cpp.html
Eclipse for C++ - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/eclipse.html
Microsoft Visual C++ (Visual Studio 2003) - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/msvc.html
Microsoft Visual C++ (Visual Studio 2005) - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/msvc8.html
Open Watcom - http://www.tecgraf.puc-rio.br/iup/en/ide_guide/owc.html

C++ BuilderX IDE Project Options Guide

http://www.borland.com/products/downloads/download_cbuilderx.html

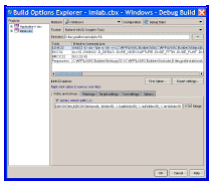
Borland C++ Builder X is an Integrated Development Environment (IDE) for Java and C/C++ languages. It can use several sets of compilers, including the Borland command line compilers version 5.6.

It also has many features, with the Borland name behind it. Its download is free. To use IUP with C++BuilderX you will need to download the "bc56" binaries in the download page.

After unpacking the file in your conputer, you must create a new Project for a "New GUI Application" and configure your Project Options. In the Project Build Options Explorer dialog there are 3 important places:

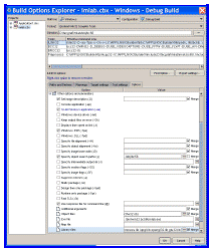
- In the Tools list, click on ILINK32. Then below select the Path and Defines tab - there you are going to add the path of the libraries you use, for example:

```
.\lib\bc56;..\..\iup\lib\bc56;..\..\cd\lib\bc56;..\..\im\lib\bc56
```



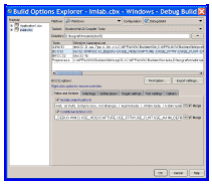
- In the same ILINK32 options, in the tab Options, select Other Options and Parameters, then Library files - there you are going to list the libraries, for example:

```
cw32.lib import32.lib vfw32.lib comctl32.lib iup.lib iupcontrols.lib cd.lib iupcd.lib im.lib im_capture.lib im_avi.lib im_process.lib iupgl.lib opengl32.lib glu32.lib
```



- In the Tools list, click on IBCC32. Then below select the Path and Defines tab - there you are going to list the include path, for example:

```
..\include;..\..\iup\include;..\..\cd\include;..\..\im\include
```

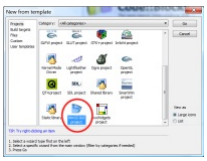


Code Blocks Project Properties Guide

http://www.codeblocks.org/

This guide was built using Code Blocks 8.02 IDE in Windows (but similar configuration can be applied for Linux).

To create a new project go to the menu "File / New / Project" and select "Win32 GUI project":

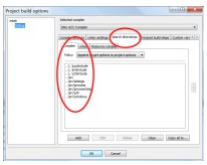


You can use several compilers, for this tutorial we will choose the MingW3 compiler. Just use the respective IUP binaries package: "mingw3".

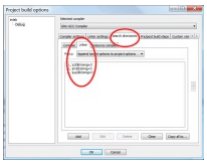
Then remove the automatically added files and add your files to the projet workspace.

After creating the project you must configure it to find the IUP includes and libraries. Go the menu "Project / Build Options".

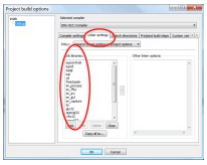
To configure the include files location go to "Search Directories" then in Compiler add the paths you need:



To configure the library files location go to "Search Directories" then in Compiler add the paths you need:



To add the library files go to "Linker Settings" then in "Link libraries" add the files you need:



Dev-C++ IDE Project Options Guide

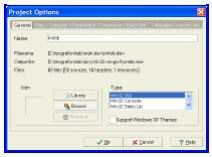
http://www.bloodshed.net/devcpp.html

"Bloodshed Dev-C++ is a full-featured Integrated Development Environment (IDE) for the C/C++ programming language. It uses Mingw port of GCC (GNU Compiler Collection) as it's compiler. Dev-C++ can also be used in combination with Cygwin or any other GCC based compiler."

It has many features, and integrated debug and it is free! To use IUP with Dev-C++ you will need to download the "mingw3" binaries in the download page.

After unpacking the file in your conputer, you must create a new Project and configure your Project Options. In the Project Options dialog there are 3 important places:

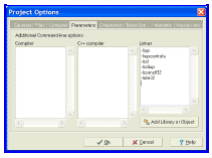
- General / Type - you can configure Win32 GUI or Win32 Console, but if you set to console it will always create a console screen behind your window when the program starts. Do not select "Support Windows XP Themes".



- Parameters / Linker - where you are going to list the libraries you use, for example:

```
-liup
-liupcontrols
-lcd
-liupcd
-lcomctl32
-lole32
-lgdi32 (if Win32 Console)
-lcomdlg32 (if Win32 Console)
```

In this configuration you are using also the additional library of Controls that uses the CD library, also available at the download page.

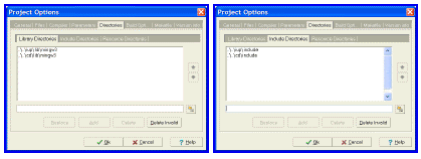


- Directories / Library Directories and Include Directories - where you are going to list the include path, for example:

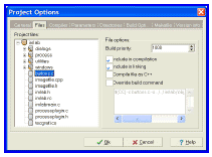
```
..\..\iup\lib\mingw3
..\..\cd\lib\mingw3
or
c:\tecgraf\iup\lib\mingw3
c:\tecgraf\cd\lib\mingw3
```

And:

```
..\..\iup\include
..\..\cd\include
or
c:\tecgraf\iup\include
c:\tecgraf\cd\include
```



In some cases the IDE may force the compilation of C files as C++. If do not want that then uncheck the option in the settings for each file. Still in the Project Options dialog, in the Files tab, select the file and uncheck "Compile File as C++".

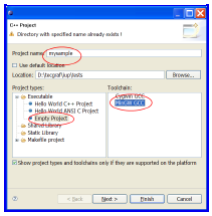


Eclipse for C++ Project Properties Guide

http://www.eclipse.org/

This guide was built using Eclipse 3.3 IDE for C/C++ Developers in Windows (but similar configuration can be applied for Linux).

To create a new project go to the menu "File / New / C or C++ Project":

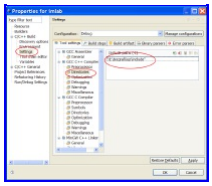


You can use the MingW3 or Cygwin compiler. Just use the respective IUP binaries package: "mingw3" or "gcc3".

Then add your files to the projet folder if they are not already there.

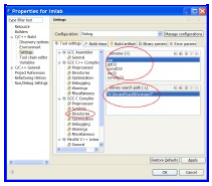
After creating the project you must configure it to find the IUP includes and libraries.

Go the menu "Project / Properties", then to configure the include files location select "GCC C Compiler / Directories" in the left tree, then add the list of folders in "Include Paths".



Be aware that you will have to repeat the configuration for the C++ compiler.

To configure the library files location select "MinGW C++ Linker / Libraries" in the left tree, then add the list of folders in "Library Search Path" and add the add the list of folders in "Libraries".



OpenWatcom C++ IDE Project Options Guide

http://www.openwatcom.org/

Open Watcom is an Integrated Development Environment (IDE) for Fortran and C/C++ languages using the Watcom compilers.

"It is a joint effort between SciTech Software Inc, Sybase and the Open Source development community to maintain and enhance the Watcom C/C++ and Fortran cross compilers and tools. An Open Source license from Sybase allows free commercial and non-commercial use of the Open Watcom tools."

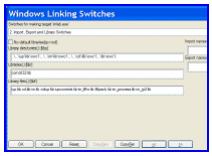
To use IUP with Open Watcom you will need to download the "owc1" binaries in the download page.

After unpacking the file in your conputer, you must create a new Project for a "Windowed Executable" and configure your Project Options. In the Project Options there are 2 important places:

- In the Windows Linking Switches dialog, select option 2. Import, Export and Library Switches. Then enter the Library directories and Library files. For example:

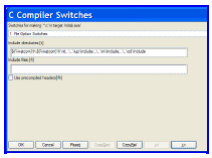
```
.\lib\owc1;..\..\iup\lib\owc1;..\..\cd\lib\owc1;..\..\im\lib\owc1
```

```
comctl32.lib iup.lib iupcontrols.lib cd.lib iupcd.lib im.lib im_process.lib iupgl.lib opengl32.lib glu32.lib
```



- In the C Compiler Switches dialog, select 1. File Option Switches. Then enter the include path, for example:

```
..\include;..\..\iup\include;..\..\cd\include;..\..\im\include
```

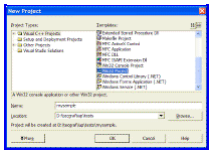


Visual C++ 7 IDE Project Properties Guide

http://msdn2.microsoft.com/en-us/vstudio/aa700867.aspx

This guide was built using Microsoft Visual Studio .NET 2003, which includes Visual C++ 7.1.

To create a new project go to the menu "File / New / Project":



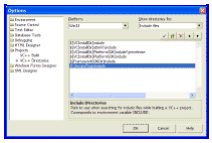
Select "Win32 Project" on the Templates. Before finishing the Wizard, select "Application Settings". Mark "Windows application" and "Empty project".

You can also create a "Console application", and whenever you execute your application a text console will also be displayed. But this is a very useful situation so you can the use standard C printf function to display textual information for debugging purposes.

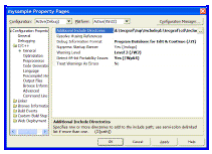
Then add your files in the menu "Project / Add New Item" or "Project / Add Existing Item".

After creating the project you must configure it to find the IUP includes and libraries. In Visual Studio there are two places where you can do this.

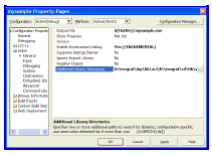
One is in the menu "Tools / Options", then select "Project / Visual C++ Directories". Select "Include Files" or "Library Files" in "Show directories for:". In this dialog you will configure parameters that will affect all the projects you open.



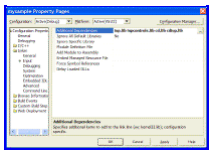
Or you can configure the parameters only for the project you created. In this case go the menu "Project / Properties". To configure the include files location select "C/C++ / General" in the left tree, then write the list of folders separated by ";" in "Additional Include Directories".



To configure the library files location select "Linker / General" in the left tree, then write the list of folders separated by ";" in "Additional Library Directories".



Now you must add the libraries you use. In this same dialog, select "Linker / Input" in the left tree, then write the list of files separated by spaces " " in "Additional Dependencies".

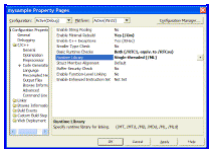


In this sample configuration the project is using the additional library of Controls that uses the CD library, also available at the download page.

When you build the project the Visual C++ linker will display the following message:

```
LINK : warning LNK4098: defaultlib 'LIBC' conflicts with use of other libs; use /NODEFAULTLIB:library
```

The default configuration use the C run time library with debug information, and IUP uses the C run time library without debug information. You can simply ignore this warning or change your project properties in "C/C++ / Code Generation" in the left tree, then change "Run Time Library" to "Single Threaded (/ML)".



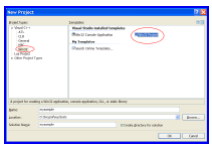
If you want to use multithreading then you must use the DLL version of the IUP libraries. They are built with the "Multi-threaded DLL (/MD)" option. Or you must rebuild the libraries with your own parameters.

Visual C++ 8 IDE Project Properties Guide

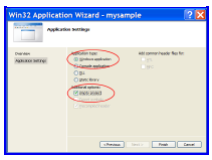
http://msdn2.microsoft.com/en-us/vstudio/default.aspx
http://msdn.microsoft.com/vstudio/express/downloads/ (free version)

This guide was built using Microsoft Visual Studio 2005, which includes Visual C++ 8. Also works for Visual Studio Express Edition.

To create a new project go to the menu "File / New / Project":



Select "Win32 Project" on the Templates. Before finishing the Wizard, select "Application Settings". Mark "Windows application" and "Empty project".

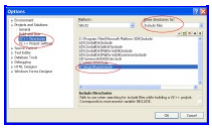


You can also create a "Console application", and whenever you execute your application a text console will also be displayed. This is a very useful situation so you can the use standard C printf functions to display textual information for debugging purposes.

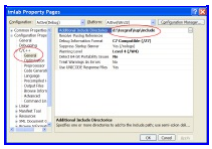
Then add your files in the menu "Project / Add New Item" or "Project / Add Existing Item".

After creating the project you must configure it to find the IUP includes and libraries. In Visual Studio there are two places where you can do this.

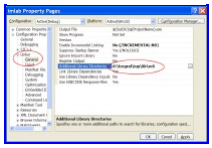
One is in the menu "Tools / Options", then select "Project and Solutions / Visual C++ Directories". Select "Include Files" or "Library Files" in "Show directories for:". In this dialog you will configure parameters that will affect all the projects you open.



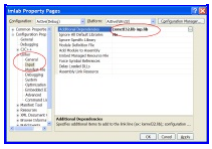
Or you can configure the parameters only for the project you created. In this case go the menu "Project / Properties". To configure the include files location select "Configuration Properties / C/C++ / General" in the left tree, then write the list of folders separated by ";" in "Additional Include Directories".



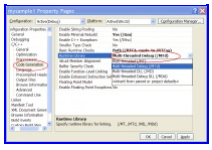
To configure the library files location select "Configuration Properties / Linker / General" in the left tree, then write the list of folders separated by ";" in "Additional Library Directories".



Now you must add the libraries you use. In this same dialog, select "Configuration Properties / Linker / Input" in the left tree, then write the list of files separated by spaces " " in "Additional Dependencies".



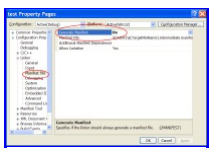
The default configuration use the C run time library with debug information and in a DLL. The standard IUP binary distribution has two packages for Visual Studio 2005 (or Visual C++ 8). Both do not have debug information, but this could be ignored even if a warning appears in the Output log. To change your project properties go to "Configuration Properties / C/C++ / Code Generation" in the left tree, then change "Run Time Library" to match the IUP binary package you are using.



The "vc8" package includes static libraries without debug information. So to match this package configuration you should select "Multi-threaded (/MT)".

The "dll8" package includes dynamic libraries without debug information. So to match this package configuration you should select "Multi-threaded DLL (/MD)".

When using the "iup.manifest" from "iup.rc", configure the linker properties of your project to do NOT generate a manifest file or the Windows Visual Styles won't work.

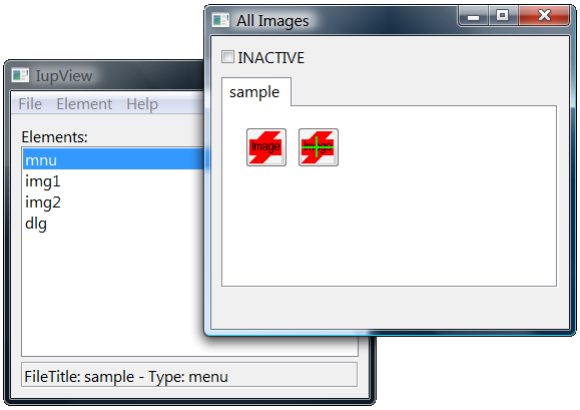


Tools Executables

IupView

The **IupView** application can be used to test LED files, load images to IupImage, or save IupImage as several formats, display all images and test them when disabled, display dialogs and popup

menus. The **IupView** application is available in the distribution files source code and pre-compiled binaries at the Download pages.



It can also be used from the command line to perform the image conversion from image files to source code that creates an **IupImage** (since 3.9).

```
iupview [-h] [-t type] [-o out_file] in_files
  IUP version: 3.9
  Converts image files to source code that creates an IupImage.
  Can pack several files in a single output file.
  Each image will correspond to a function called load_image_,
  where  is the file name of the input image without path.
  -h            print this help
  -t            output format, can be LED, LUA or C (default: C)
  -o out_file   place output in file  (default: images.c)
```

For example:

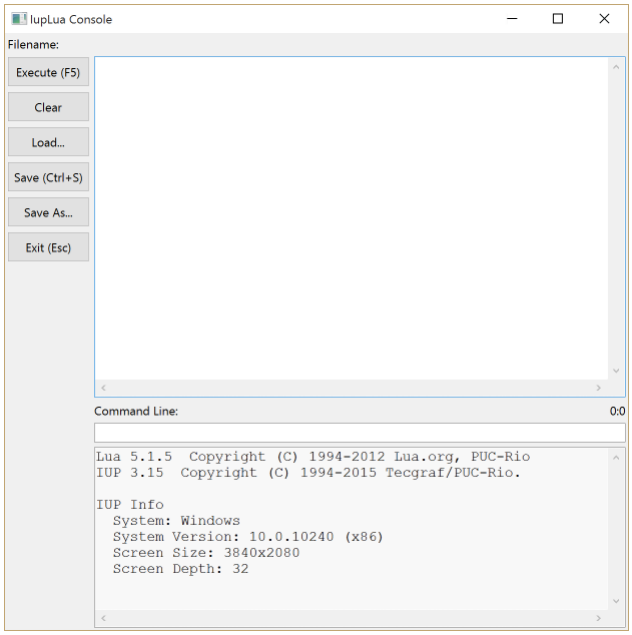
```
> iupview -t LUA EditCut.png

> cat images.c
function load_image_EditCut_png()
  local EditCut_png = iup.imagergba
  {
    width = 16,
    height = 16,
    pixels = {
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 169, 178, 191, 139, 155, 166, 178, 228, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 47
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 147, 157, 172, 52, 140, 150, 163, 254, 122, 133, 153, 25, 255, 255, 255, 0, 246, 247, 248, 13
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 146, 155, 169, 250, 147, 156, 171, 163, 255, 255, 255, 15, 230, 232, 236, 2
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 169, 177, 190, 242, 138, 147, 161, 249, 200, 204, 209, 128, 193, 198, 206,
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 161, 170, 184, 179, 139, 148, 162, 254, 141, 150, 161, 191, 119, 128, 142,
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 137, 145, 161, 65, 116, 128, 142, 254, 99, 110, 126, 254, 85, 93, 105, 253,
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 96, 110, 132, 238, 86, 96, 113, 255, 84, 102, 134, 252, 7
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 91, 136, 238, 45, 88, 137, 235, 179, 84, 131, 224, 249, 75, 118, 202, 173, 76, 119, 205, 251,
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 88, 135, 234, 49, 89, 140, 238, 243, 83, 128, 220, 247, 59, 91, 158, 244, 53, 82, 142, 221, 70, 110, 190, 236,
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 86, 135, 231, 176, 86, 134, 229, 255, 65, 102, 176, 255, 53, 82, 142, 248, 53, 82, 141, 180, 75, 116, 202, 250,
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 85, 132, 226, 250, 82, 129, 222, 49, 66, 101, 178, 48, 51, 80, 139, 250, 53, 82, 141, 18, 80, 123, 213, 250, 77
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 83, 130, 222, 250, 79, 118, 210, 48, 62, 99, 172, 98, 49, 76, 132, 250, 255, 255, 255, 0, 75, 116, 199, 205, 62
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 78, 121, 209, 206, 63, 98, 171, 255, 51, 80, 138, 255, 39, 54, 87, 227, 22, 18, 14, 3, 63, 100, 172, 89, 42, 59
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 67, 104, 180, 88, 54, 84, 146, 227, 40, 58, 93, 238, 49, 74, 124, 110, 255, 255, 255, 0, 255, 255, 255, 0, 255,
      255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255, 255, 255, 0, 255
    },
  }
  return EditCut_png
end
```

IupLua Console

The **IupLuaConsole** can load and execute Lua scripts using the IupLua binding. Lua print calls are output in a IupMultiline. The executable package also includes the CD, IM, LuaFileSystem and LuaGL libraries.

The **IupLuaConsole** application is available in the distribution files source code and pre-compiled binaries at the Download pages. The executable from the distribution package in UNIX needs that the LD_LIBRARY_PATH (DYLD_LIBRARY_PATH in MacOSX) environment variable must contains the executable folder, for example:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/tecgraf/iup3/bin/$TEC_UNAME
```



LED Compiler for C

Description

The LED compiler (**ledc**) generates a C module from one or more LED files. The C module exports only one function, which builds the IUP interface described in the LED files. Running this function is equivalent to calling the IupLoad function over the original LED files.

One advantage of using the compiler is that it allows the application to be independent from LED files during its execution. Since the interface description is inside the executable file, there is no need to worry about locating the configuration files.

Another advantage is that **ledc** performs a stricter verification than IUP's internal parser. This makes error detection in LED files easier.

Finally, running the function generated by the compiler is faster than reading the corresponding LED file with IupLoad, since the parsing step of the LED file is transferred from execution to compilation. However, creating the IUP elements described in LED takes most of the execution time of the IupLoad function, so the gain in efficiency may not be very significant.

Usage

ledc [-v] [-c] [-f funcname] [-o file] files

-v	shows **ledc**'s version number
-c	does not generate code, just checks for errors in the LED files
-f funcname	uses <funcname> as the name of the generated exported function (default: led_load)
-o file	uses <file> as the name of the generated file (default: led.c)

Error Messages

Several warnings and error messages might be generated during compilation. Errors abort the compilation. The messages can be the following:

warning: undeclared control *name* (argument *number*)
The *name* name was used as an argument where a IUP element was expected, but no element with this name was previously declared.

warning: string expected (argument *number*)
A name (callback?) was passed as a parameter for a string-type argument.

warning: callback expected (argument *number*)
A string was passed as a parameter for a callback-type argument.

warning: unknown control *name* used
An unknown element, called *name*, was used. The compiler assumes the element's creation function is called Iup*Name*, with *name* capitalized, and assumes the arguments' types based on what was passed on LED.

warning: *elem* declared without a name
An *elem*-type element was declared without being associated to any name. This declaration creates the element, but it will not be accessible, so it cannot be used.

element *name* already used in line *number*
The *name* element was already used in line *number*. In IUP, the same element cannot have more than one parent.

too few arguments for *name*
The *name* element expects more arguments than those already passed.

too many arguments for *name*
The *name* element expects less arguments than those passed.

name is not a valid child
The *name* element cannot be used as a parameter in this case. This happens when trying to insert an image into a vbox, for instance.

control expected (argument *number*)
A string was passed as a parameter for an element-type argument.

string expected (argument *number*)
An element was passed as a parameter for a string-type argument.

number expected (argument *number*)
An element or a string was passed as a parameter for a number-type argument.

callback expected (argument *number*)
An element was passed as a parameter for a callback-type argument.

hotkeys not implemented
Even though it is a LED word reserved to an element, it is not implemented.

Complete Samples

Standard Controls

The following example creates a dialog with virtually all of IUP standard elements as well as some variations of them, with some attributes changed. The same example is implemented in C, LED and Lua. The C code is ready to compile. The LED code can be loaded and viewed in the IupView application. The Lua code can be loaded and executed in the **IupLua** standalone application.

in C	in LED	in IupLua
sample.c	sample.led	sample.lua

You can see the results in Windows, Motif and GTK on the Sample Results.

All Samples

The IUP samples are spread in the documentation. Each control, dialog, menu has its own set of examples in C, LED and Lua.

Browse for Example Files

iupglcap

This application uses IUP and OpenGL to create a window with two canvases and draw a video capture image into one canvas. A processed image can be displayed in the second canvas. It can also process frames from a video file. It is very useful for Computer Vision courses. You can download the source code here: iupglcap.zip. You will also need to download IUP, CD and IM libraries for the compiler you use.

External Samples

The CD and IM libraries have samples that use IUP, check in their documentation.

Some freely available applications also use IUP:

IMLAB - Image Processing Laboratory

EdPatt - Pattern Editor

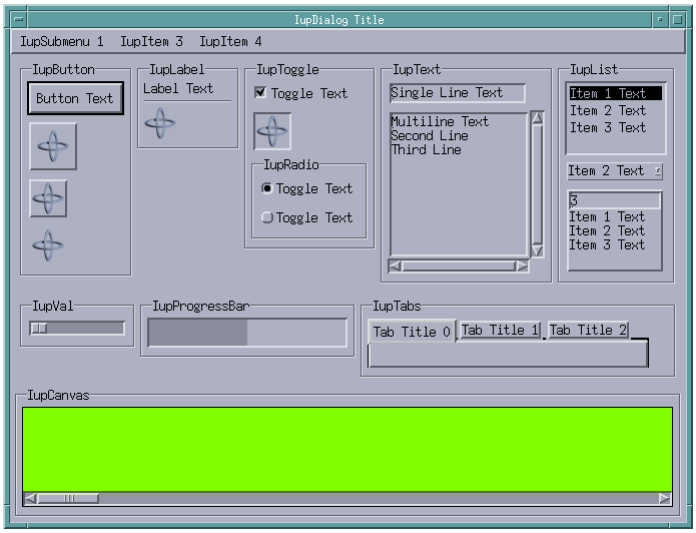
Ftool - Two-dimensional Frame Analysis Tool

The Lua for Windows distribution is a 'batteries included environment' for the Lua scripting language on Windows, that also includes LuaGL, IUP, CD and IM.

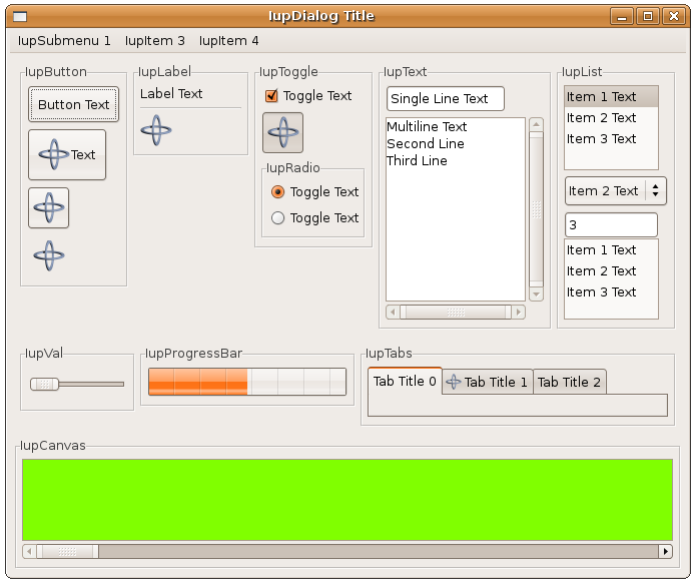
Sample Results (1)

The following screenshots shows the sample.c results. See also the same sample changing the dialog BACKGROUND, the dialog BGCOLOR and the children BGCOLOR.

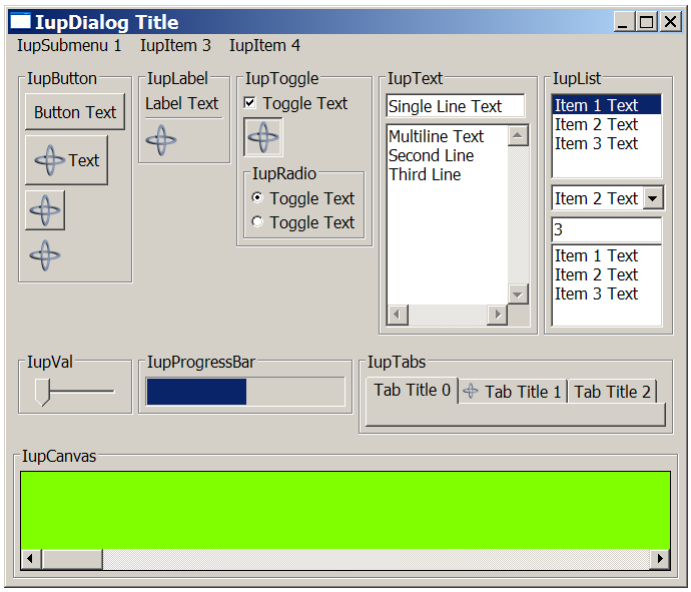
Motif in MWM



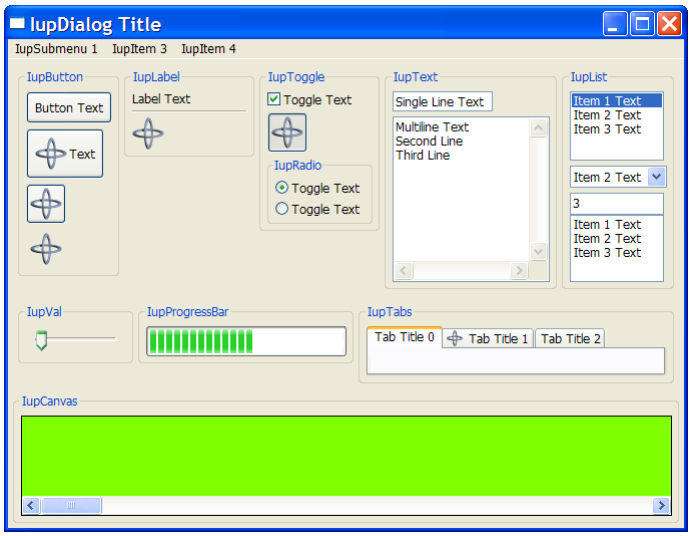
GTK in Gnome



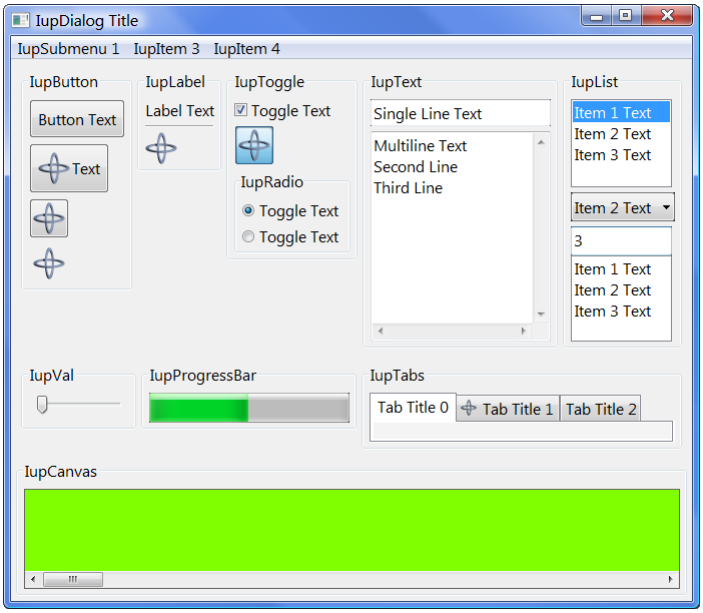
Windows Classic



Windows with Visual Styles



Windows Vista

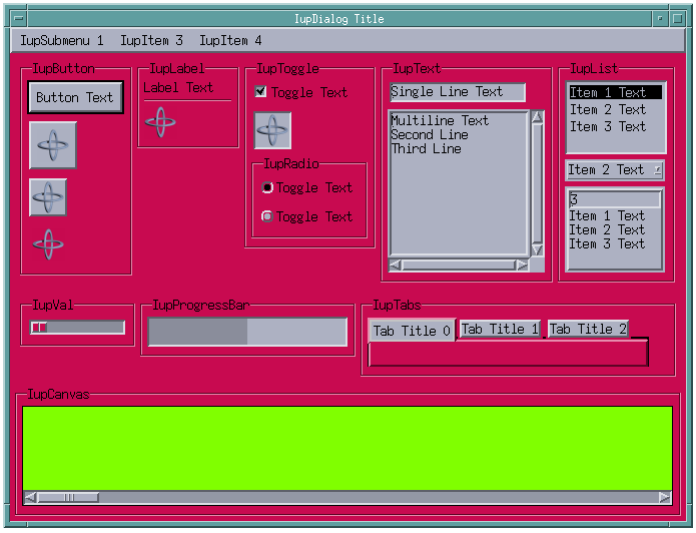


Sample Results (2)

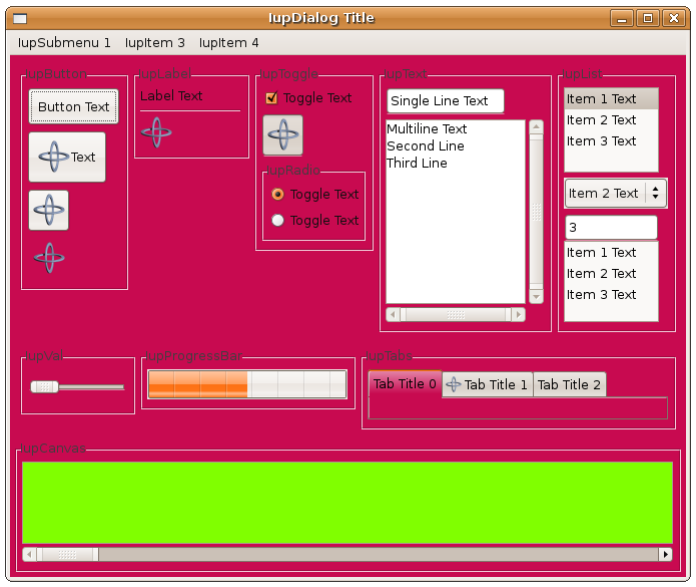
The following screenshots shows the sample.c results when the dialog BACKGROUND attribute is changed. See also the same sample with normal background, the dialog BGCOLOR and the children BGCOLOR.

Notice that the dialog BACKGROUND attribute affects only the background of the dialog, the background of each control is preserved except for the ones that the background is transparent.

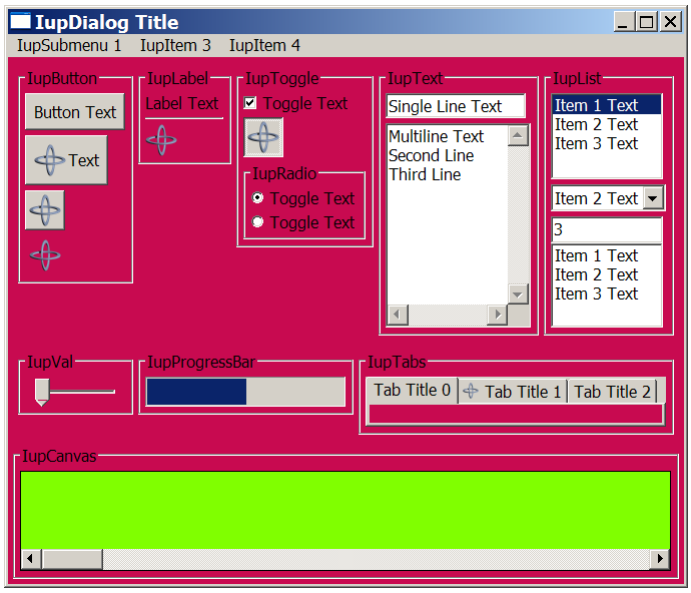
Motif in MWM



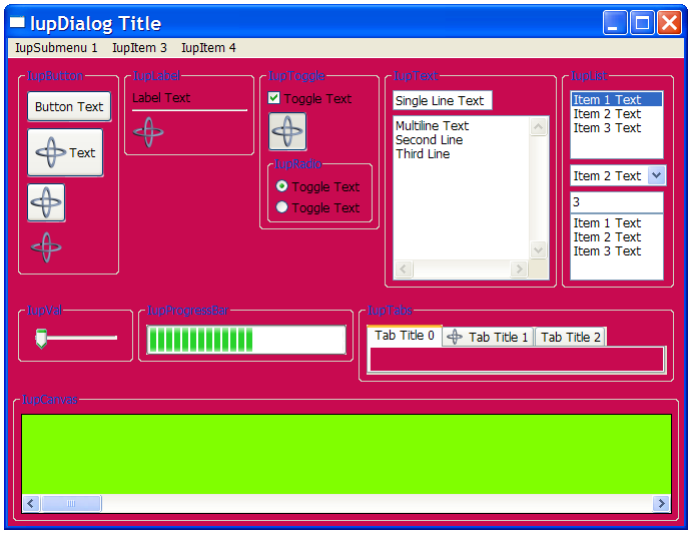
GTK in Gnome



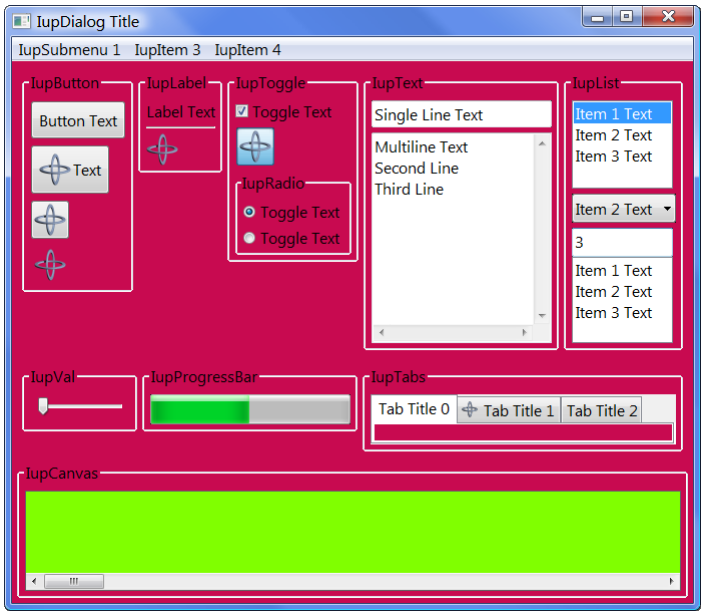
Windows Classic



Windows with Visual Styles



Windows Vista

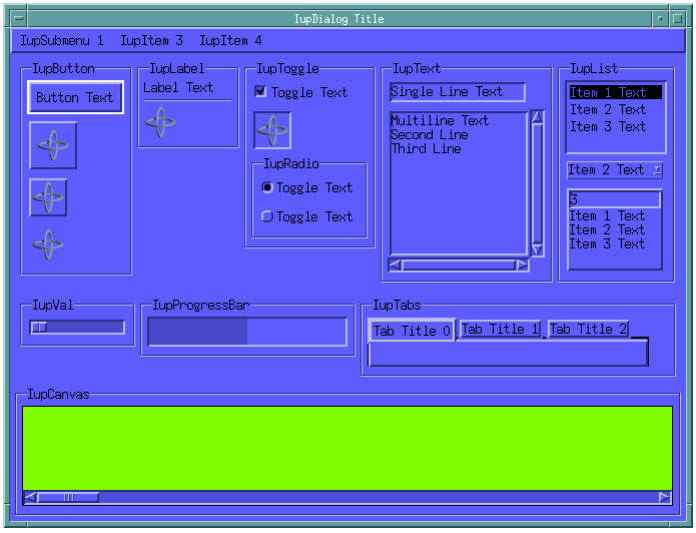


Sample Results (3)

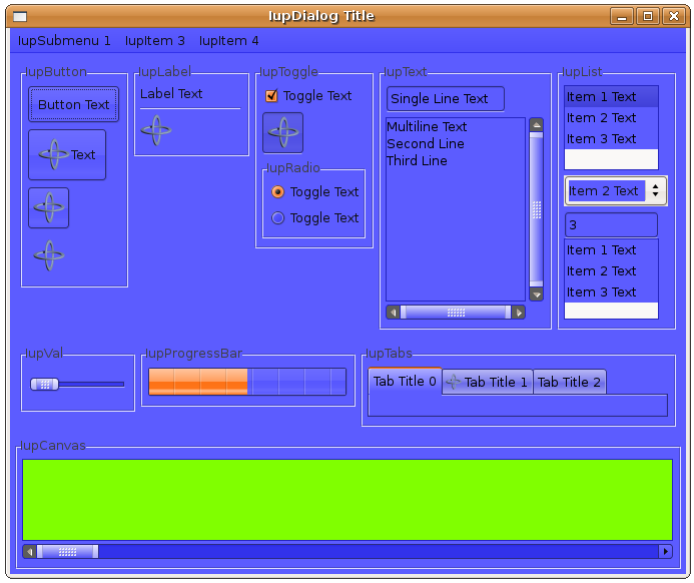
The following screenshots shows the sample.c results when the dialog BGCOLOR attribute is changed. See also the same sample with normal background, changing the dialog BACKGROUND and the children BGCOLOR.

Since BGCOLOR is an inheritable attribute changing it at the dialog affects all controls. And notice that on Windows the BGCOLOR is ignored for several controls.

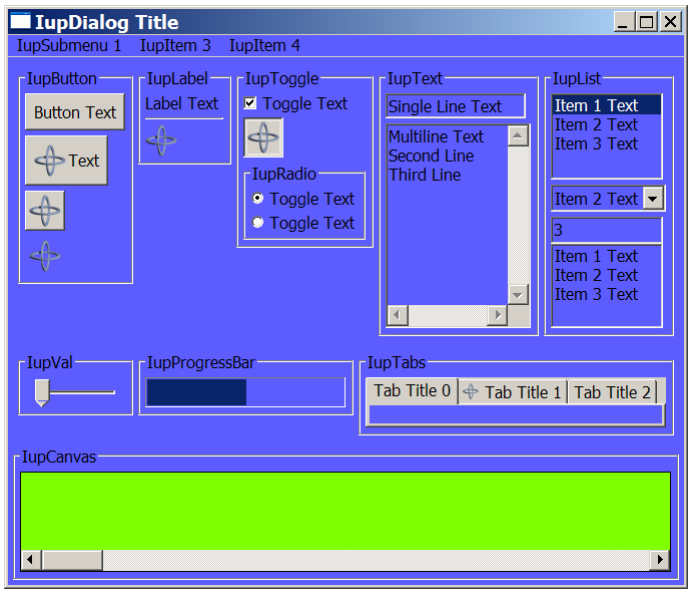
Motif in MWM



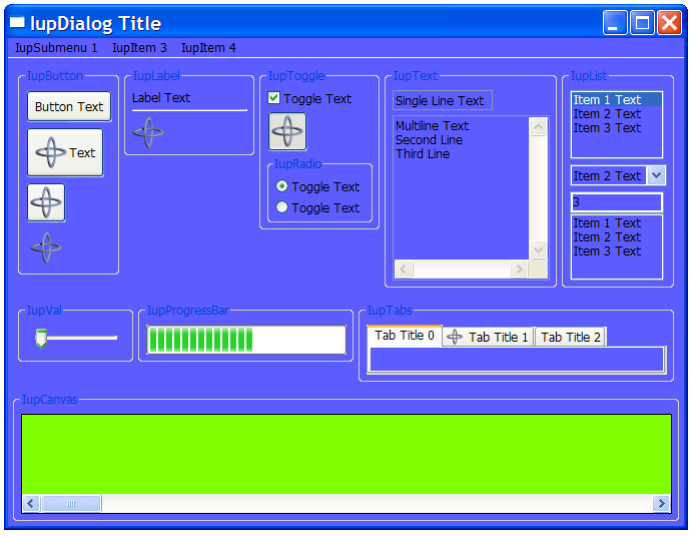
GTK in Gnome



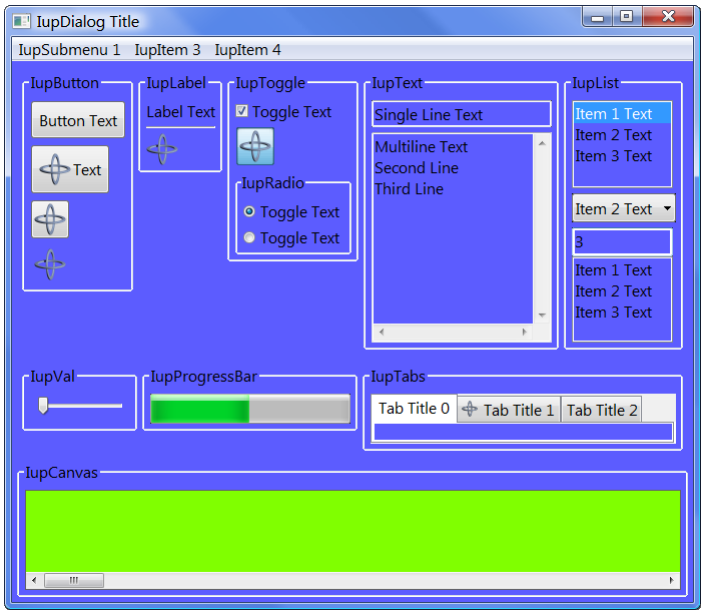
Windows Classic



Windows with Visual Styles



Windows Vista

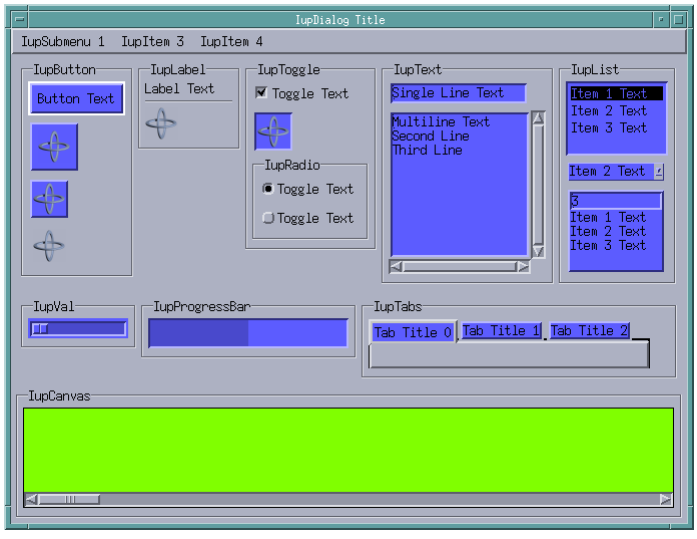


Sample Results (4)

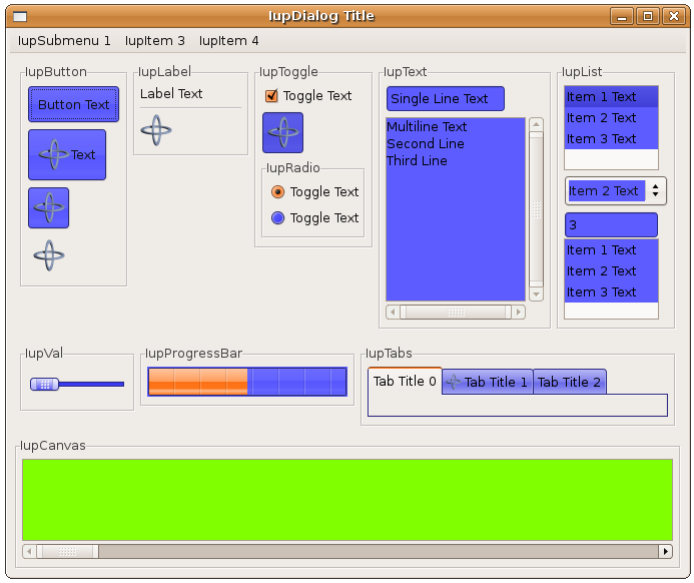
The following screenshots shows the sample.c results when the BGCOLOR attribute of the dialog children is changed, but NOT the dialog. See also the same sample with normal background, changing the dialog BACKGROUND and the dialog BGCOLOR.

In this case, the BGCOLOR attribute affects only the controls. Also notice that the transparent area of the controls are not affected. And notice that on Windows the BGCOLOR is ignored for several controls.

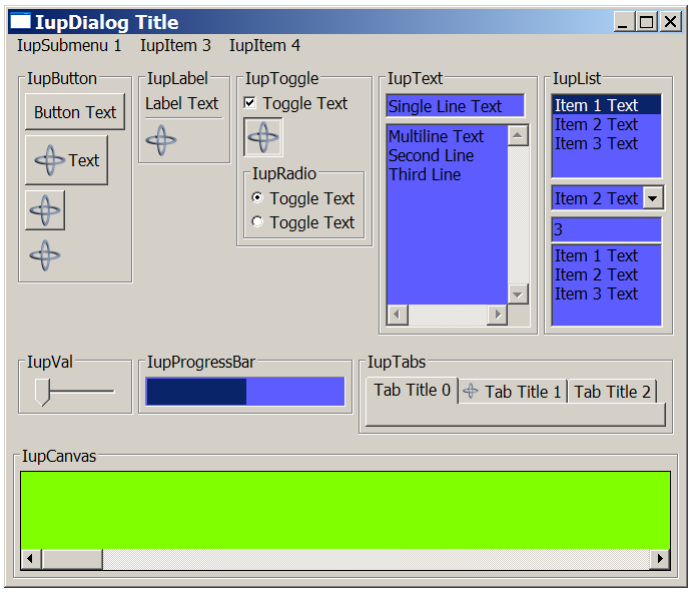
Motif in MWM



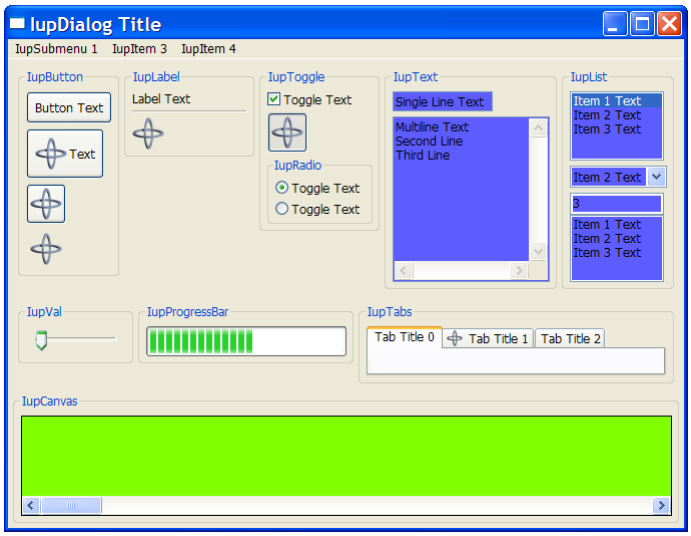
GTK in Gnome



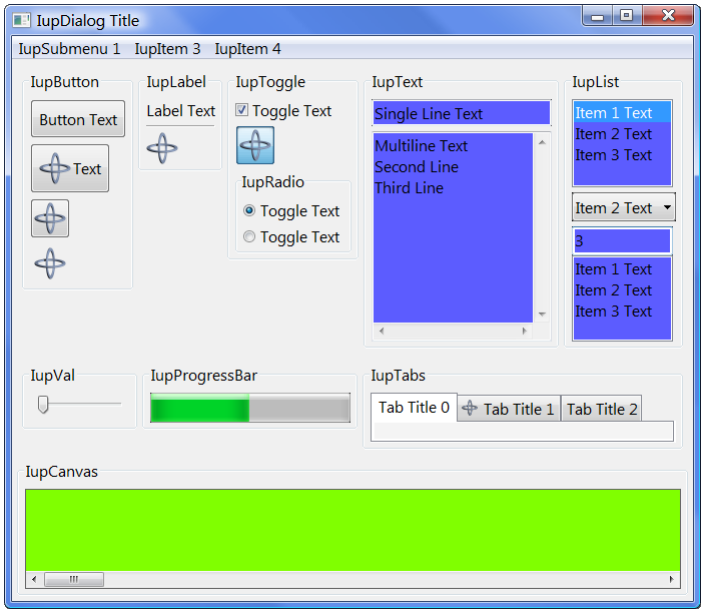
Windows Classic



Windows with Visual Styles



Windows Vista



Lua Binding

Overview

All the IUP functions are available in Lua, with a few exceptions. We call it **IUPLua**. To use them the general application will do require"iuplua", and require"iupluaxxxx" to all other secondary libraries that are needed. The functions and definitions will be available under the table "iup" using the following name rules:

```
iupXxx  -> iup.Xxx    (for functions)
IUP_XXX -> iup.XXX    (for definitions)
```

All the metatables have the "tostring" metamethod implemented to help debugging.

Also the functions which receive values by reference in C were modified. Generally, the values of parameters that would have their values modified are now returned by the function in the same order.

Notice that, as opposed to C, in which enumeration flags are combined with the bitwise operator OR, in Lua the flags are added arithmetically.

In Lua all parameters are checked and a Lua error is emitted when the check fails.

All the objects are NOT garbage collected by the Lua garbage collector, you must manually call **iup.Destroy** or elem:**destroy**, if you would like to destroy an element.

In Iup additional features were created for the Lua Binding using the metamethods. Attributes and callbacks can be set and get in a much more natural way:

```
IupSetAttribute(label, "TITLE", "test")   >>   label.title = "test"            (names are in lower case)
title = IupGetAttribute(label, "TITLE")   >>   title = label.title

IupSetCallback(button, "ACTION", button_action_cb);   >>    function button:action() ... end
```

Also the element constructors were changed so you can use tables to initialize their parameters and attributes:

```
IupButton("test")         >>    iup.button{title = "test", alignment="acenter"}
IupHbox(bt1, bt2, NULL)   >>    iup.hbox{bt1, bt2, margin="10x10"}
```

Lua was created after **LED**, so that's why **LED** exists. Since we have many application still using LED, its support will continue in IUP. Today **IupLua** completely replaces the LED functionality and adds much more.>

The distribution files include an executable called **iuplua51**, that you can use to test your Lua code. It has support for all the additional controls, for IM, CD and OpenGL calls. It is available at the Download.

IupLua Initialization

Lua 5.1 "require" can be used for all the **IupLua** libraries. You can use **require**"**iuplua**" and so on, but the LUA_CPATH must also contains at least the following:

```
"./lib?51.so;"    [in UNIX]

".\\?51.dll;"     [in Windows]
```

The LuaBinaries distribution already includes these modifications on the default search path.

The simplest form **require**"**iup**" and so on, can not be used because there are IUP dynamic libraries with names that will conflict with the names used by **require** during search.

Additionally you can statically link the **IupLua** libraries, but you must call the initialization functions manually. The **iuplua_open** function is declared in the header file **iuplua.h**, see the example below:

```
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>
#include <iuplua.h>

void main(void)
{
  lua_State *L = lua_open();

  luaopen_string(L);
  luaopen_math(L);
  luaopen_io(L);

  iuplua_open(L);
```

```
  lua_dofile("myprog.lua");

  lua_close(L);
}
```

When using **Lua** the Iup initialization functions, **IupOpen**, **IupControlsOpen** and others, are not necessary. The initialization is automatically done inside the respective **IupLua** initialization function.

To use IUP inside coroutines, define the global attribute "IUPLUA_THREADED".

Embedding Lua files in the Application Executable

The Lua files are dynamically loaded and must be sent together with the application's executable. However, this often becomes an inconvenience. To deal with it, there is the **LuaC** compiler that creates a C module from the Lua contents. For example:

```
luac -o myfile.lo myfile.lua
lua bin2c.lua myfile.lo > myfile.loh
```

In C, you can use a define to interchanged the use of .LOH files:

```
#ifdef _DEBUG
  ret_val = iuplua_dofile("myfile.lua");
#else
#include "myfile.loh"
#endif
```

This does not work when using LuaJIT. To be able to do that, use Lua files directly as strings:

```
lua bin2c.lua myfile.lua > myfile.lh
```

In C, simply include the .LH files:

```
#include "myfile.lh"
```

More Information

Steve Donovan wrote a very nice "A Basic Guide to using IupLua" that was included in Lua for Windows. It is now available as part of the IUP documentation.

The slides for **"Tecgraf Development Tools: IUP, CD and IM"** presented at the Lua Workshop 2009 are also available for Download [iupcdim_wlua2009_en.pdf].

A Basic Guide to using IupLua

by Steve Donovan

IupLua is a cross-platform kit for creating GUI applications in Lua. There are particularly powerful facilities for getting user input that don't require complicated coding, so it is particularly good for utility scripts.

Attributes are an important concept in IUP. You set and get them just like table fields, but they are different from fields in several crucial ways. First, case is not significant, `SIZE` is just as good as `size` (but try to be consistent!). Second, writing to a non-existent attribute will *not* give you an error, so proof-read carefully. Third, writing to an attribute often causes some action; e.g the `visible` attribute of controls can be used to hide them. It is best to think of them as a special kind of function call.

Functions which create IupLua objects (i.e. *constructors*) take tables as arguments. Lua allows you to drop the usual parentheses in such a case, but remember that something like `iup.fill{}` is not the same as `iup.fill()`; it is actually short for `iup.fill({})`. A Lua table can contain an array-like part (just items separated by commas) and a map-like part (attribute-value pairs); the convention is to put the array part first, and separate the map part from it with a semicolon. (See Attributes/Guide/IupLua in the Manual for a good discussion.)

All the examples presented here and some utilities can be found at the "misc" folder in the IupLua examples.

Simple Output

Even simple scripts need to give the user some feedback. Otherwise people get anxious and start worrying if their files really have been backed up, for example. This is easy in IUPLua, and takes exactly one line. Note that all IUP scripts must at least have a `require 'iuplua'` statement at the begining:

```
require( "iuplua" )

iup.Message('YourApp','Finished Successfully!')
```

Of course, many operations require confirmation from the user. `iup.Alarm` is designed for this:

```
require( "iuplua" )

b = iup.Alarm("IupAlarm Example", "File not saved! Save it now?" ,"Yes" ,"No" ,"Cancel")

-- Shows a message for each selected button
if b == 1 then
  iup.Message("Save file", "File saved sucessfully - leaving program")
elseif b == 2 then
  iup.Message("Save file", "File not saved - leaving program anyway")
elseif b == 3 then
  iup.Message("Save file", "Operation canceled")
end
```

Like `iup.Message`, the first parameter appears in the title bar of the dialog box, the second parameter appears above the buttons, but `iup.Alarm` allows you to specify a number of buttons. The return code will then tell you which button has been pressed, starting at 1 (which is always the Lua way.)

Simple Input

Asking for a Filename

The most common thing an interactive script will require from a user is a file, or set of files. For simple cases, `iup.GetFile` will do the job:

```
require( "iuplua" )

f, err = iup.GetFile("*.txt")
if err == 1 then
  iup.Message("New file", f)
elseif err == 0 then
  iup.Message("File already exists", f)
elseif err == -1 then
```

```
  iup.Message("IupFileDlg", "Operation canceled")
end
```

This will present you with the standard Windows File Open dialog box, and allow you to either choose a filename, or cancel the operation. Notice that this function returns two values, the filename and a code. The code will tell you whether the file does not exist yet (if for instance you typed a new filename into the file dialog box.)

Asking for Multiline Text

The simplest way of getting general text is to use `iup.GetText`:

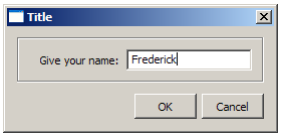
```
require 'iuplua'

res = iup.GetText("Give me your name","")

if res ~= "" then
    iup.Message("Thanks!",res)
end
```

Using this dialog, you can enter as many lines as you like, and press OK.

Asking for a Single String, or Number

A better option for asking for a single string is the very versatile `iup.GetParam`:

```
require( "iuplua" )
require( "iupluacontrols" )

res, name = iup.GetParam("Title", nil,
    "Give your name: %s\n","")

iup.Message("Hello!",name)
```



This has two advantages over plain `GetText`; you can give a prompt line, and you can press Enter after entering text.

The `%s` code requires some explanation. Although you might at first think it is a C-style formating code, as you would use in `string.format`, it actually describes how the value is going to edited; `%s` here merely means that a regular text box is used; if you had used `%m`, then a multiline edit box (like that used by `iup.GetText`) would be used.

If there is a limited set of choices, then the `%l` format is useful:

```
res, prof = iup.GetParam("Title", nil,
    "Give your profession: %l|Teacher|Explorer|Engineer|\n",0)
```

Note the `|item1|item2|...|` list after the `%l` format; these are the choices presented to the user. The initial value you give it, and the value you receive from it, are going to be an index into this list of choices. Somewhat confusingly, they start at 0 (which is not the Lua way!) So in this case, `0` means that 'Teacher' is to be selected, and if I then selected 'Engineer', the resulting value of `prof` would be 2.

The `%i` code allows you to enter an integer value, with up/down arrows for incrementing/decrementing the value.

```
require( "iuplua" )
require( "iupluacontrols" )

res, age = iup.GetParam("Title", nil,
    "Give your age: %i\n",0)

if res ~= 0 then    -- the user cooperated!
    iup.Message("Really?",age)
end
```

Dialogs

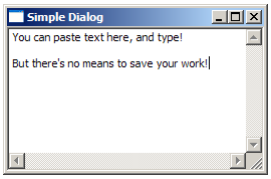
Constructing General Layouts

`GetParam` is a very versatile facility for asking for data, but it is not very interactive. In general, you want to present something back to the user that is more complicated than a simple message. Up to now we have used the predefined dialogs available to IupLua; it is now time to go beyond that and examine custom dialogs. The structure of a simple IupLua program is straightforward:

```
require 'iuplua'

text = iup.multiline{expand = "YES"}

dlg = iup.dialog{text; title="Simple Dialog",size="QUARTERxQUARTER"}

dlg:show()

iup.MainLoop()
```

A multiline edit *control* is created, and put inside a window frame with a given size, which is then made visible with the `show` method. We then enter the main loop of the application with `MainLoop`, which will only finish when the window is closed.

Controls are also windows, but without the frame and decorations of a *top-level* window; they are always meant to be inside some window frame or other *container*. We set the `expand` attribute of `multiline` to force it to use up all the available space in the frame, so that it takes its size from its container. The dialog's `size` attribute is a string of the form "XSIZExYSIZE", where sizes can be expressed as fractions of the desktop window size, in this case a quarter of the width and height. (You can of course also use numerical sizes like "100x301" but these will not always scale well on displays with different resolutions. See Attributes/Common/SIZE in the manual for these units.)



It's good to pause a moment to look at the resulting application in action; it is fully responsive and you can enter text, paste, etc. into the edit control. Common keyboard shortcuts like `Ctrl+V` and `Ctrl+C` work as expected. All this functionality comes with the windowing system you are currently using. On my system, Task Manager shows that this program uses 3.8 Meg of memory, and an instance of Notepad uses 3.3 Meg, which represents all the common code necessary to support a simple GUI application; you are not actually paying much for using IupLua at all. The equivalent C program using the Windows API would be about 150 lines, so the gain in *programmer efficiency* is tremendous!

Of course, there is not much interaction possible with such a simple program. To make a program respond to the user we define *callbacks* which the system calls when some event takes place. For example, we can put a button control in the dialog, and define its `action` callback:

```
require 'iuplua'

btn = iup.button{title = "Click me!"}

function btn:action ()
    iup.Message("Note","I have been clicked!")
    return iup.DEFAULT
end

dlg = iup.dialog{btn; title="Simple Dialog",size="QUARTERxQUARTER"}

dlg:show()

iup.MainLoop()
```

This is perfectly responsive, although not very useful! The button sizes itself to its natural size since `expand` is not set (try setting `expand` to see the button fill the whole window frame.) Callbacks usually return the special value `iup.DEFAULT`, although in IupLua this is not really necessary.

`dialog` takes only one control, so IupLua defines containers in which you can pack as many controls as you like. Here `vbox` is used to pack two buttons into the dialog vertically (To save space I'm leaving out the 'dlg:show...' common code at the bottom)

```
btn1 = iup.button{title = "Click me!"}
btn2 = iup.button{title = "and me!"}

function btn1:action ()
    iup.Message("Note","I have been clicked!")
end

function btn2:action ()
    iup.Message("Note","Me too!")
end

box = iup.vbox {btn1,btn2}

dlg = iup.dialog{box; title="Simple Dialog",size="QUARTERxQUARTER"}
```

This does the job, although the buttons are sized differently according to their contents; this program would not win any design contests! Still, you now have two commands in your application. You can actually get a more pleasing result by using a horizontal packing box (`hbox`) and specifying a non-zero gap between the buttons:

```
box = iup.hbox {btn1,btn2; gap=4}
```

You can nest boxes as much as you like, which is the way to construct more complicated layouts. Here are our horizontal buttons packed vertically with a multiline edit control:

bbox = iup.hbox {btn1,btn2; gap=4} text = iup.multiline{expand = "YES"} vbox = iup.vbox{bbox,text}

dlg = iup.dialog{vbox; title="Simple Dialog",size="QUARTERxQUARTER"}

We have effectively implemented a crude but functional toolbar:

A labeled frame can be put around a control using `iup.frame`:

```
edit1 = iup.multiline{expand="YES",value="Number 1"}
edit2 = iup.multiline{expand="YES",value="Number 2?"}

box = iup.hbox {iup.frame{edit1;Title="First"},iup.frame{edit2;Title="Second"}}
```

A useful way to present various views to a user is to put them in tabs. This places each control in a separate page, accessible through the tabbar at the top. Notice in this example that the titles of the tab pages are actually set as attributes of the *pages* through `tabtitle`. This is not one of the standard IUP controls (see Controls/Additional in the manual) so we also need to bring in the `iupluacontrols` library.

```
require( "iupluacontrols" )

edit1 = iup.multiline{expand="YES",value="Number 1",tabtitle="First"}
edit2 = iup.multiline{expand="YES",value="Number 2?",tabtitle="Second"}

tabs = iup.tabs{edit1,edit2,expand='YES'}
```

Timers and Idle Processing

Sometimes a program needs to wake up and perform some operation, such as a scheduled backup or an autosave operation. IupLua provides *timers* for this purpose. (Note at this point that there is no reason why you can't have `print` in a IupLua application; sometimes there is no better way to track what's going on. But on Windows you do have to run the program using the regular lua.exe, not wlua.exe.)

```
-- timer1.lua

require "iuplua"

timer = iup.timer{time=500}

btn = iup.button {title = "Stop",expand="YES"}

function btn:action ()
    if btn.title == "Stop" then
        timer.run = "NO"
        btn.title = "Start"
    else
        timer.run = "YES"
        btn.title = "Stop"
    end
```

```
end

function timer:action_cb()
    print 'timer!'
end

timer.run = "YES"

dlg = iup.dialog{btn; title="Timer!"}
dlg:show()
iup.MainLoop()
```

After a timer has been started by setting its `run` attribute to "YES", it will continue to call `action_cb` using the given time in milliseconds. Notice that it is important to set the timer going only after the callback has been defined. It is perfectly permissable to switch a timer off in the callback, which is how you can perform a single action after waiting for some time.

It is a well-known fact that computers spend most of the time doing very little, waiting for incredibly slow humans to type something new. However, when a computer is actually doing intense processing, users become impatient if not told about progress. If you do your lengthy processing directly, then the windows of the application become unresponsive. The proper way to organize such work is to do it when the system is *idle*.

IupLua provides a gauge control which is intended to show progress; this little program shows that even when the computer is almost completely preoccupied doing work, it is still keeping the user informed and in fact the window remains useable, although a little slow to respond.

```
-- idle1.lua
require "iuplua"
require "iupluacontrols"

function do_something ()
    for i=1,6e7 do end
end

gauge = iup.gauge{show_text="YES"}

function idle_cb()
    local value = gauge.value
    do_something()
    value = value + 0.1
    if value > 1.0 then
        value = 0.0
    end
    gauge.value = value
    return iup.DEFAULT
end

dlg = iup.dialog{gauge; title = "IupGauge"}

iup.SetIdle(idle_cb)

dlg:showxy(iup.CENTER, iup.CENTER)

iup.MainLoop()
```

Lists

It is easy to display a list of values in IupLua. The values can be directly specified in the `iup.list` constructor, like so:

```
-- list1.lua
require "iuplua"

list = iup.list {"Horses","Dogs","Pigs","Humans"; expand="YES"}

dlg = iup.dialog{list; title="Lists"}
dlg:show()
iup.MainLoop()
```

(Remember that the single argument to these constructors is just a Lua table, which you can construct in any way you choose, say by reading the values from a file.)

Tracking selection changes is straightforward:

```
function list:action(t,i,v)
    print(t,i,v)
end
```

Now, as I move the selection through the list, from the start to the finish, the output is:

```
Horses  1       1
Horses  1       0
Dogs    2       1
Dogs    2       0
Pigs    3       1
Pigs    3       0
Humans  4       1
```

So `v` is 1 if we are selecting an item, 0 if we are deselecting it; `i` is the one-based index in the list, and `t` is the actual string value. If you want to associate some other data with each value, then all you need to do is keep a table of that data and look it up using the index `i`.

To register a double-click is a little more involved. There is (as far as I can tell) no way to detect whether a double-click has happened in the `action` callback. So we track the selection manually; if two selection events for a given item happen consecutively, then that is understood to be a double-click. It ain't pretty, but it works (except perhaps for the valid case of a person wanting to double-click the same item repeatedly):

```
local lastidx,doubleclick

function on_double_click (t,i)
    print(t,i)
end

function list:action(t, i, v)
    if v ~= 0 then
        if lastidx == i and doubleclick ~= i then
            on_double_click(t,i)
            doubleclick = i
        end
        lastidx = i
    end
end
```

Once a list has been created, how does one change the contents? The answer is that the list object *behaves* like an array. For example, to fill a list with all the entries in a directory, I can use this function:

```
function fill (path)
    local i = 1
    for f in lfs.dir(path) do
        list[i] = f
        i = i + 1
    end
    list[i] = nil
end
```

Note that this does not mean that a list object *is* a table. In particular, you have to explicitly set the end of the list of elements by setting a nil value just after the end.

Trees

The most flexible way to present a hierarchy of information is a tree. A tree has a single *root*, and several *branches*. Each of these *branches* may have *leaves*, and other branches. All of these are called *nodes*. Thinking of a family tree, a node may have *child* nodes, which all share the same *parent* node.

A good example of this in everyday computer experience is a filesystem, where the leaves are files and the branches are directories. Lua tables naturally express these kind of *nested* structures easily, and in fact it is easy to present a Lua table as a tree, where array items are leaves, and the branches are named with the special field `branchname`:

```
require 'iuplua'
require 'iupluacontrols'

tree = iup.tree{}
tree.addexpanded = "NO"

list = {
  {
    "Horse",
    "Whale";
    branchname = "Mammals"
  },
  {
    "Shrimp",
    "Lobster";
    branchname = "Crustaceans"
  },
  {
    branchname = "Birds"
  },
  branchname = "Animals"
}

iup.TreeAddNodes(tree, list)

f = iup.dialog{tree; title = "Tree Test"}

f:show()

iup.MainLoop()
```

This example begins with the branches 'collapsed', and you will have to explicitly expand them with a mouse click. By default, trees are presented in their fully expanded form; try taking out the fourth line that sets the `addexpanded` attribute of the tree object. Note that branches can be empty!

Tree operations are naturally more complicated than list operations, but there is a callback which happens when a node is selected or unselected. Add this to the example:

```
function tree:selection_cb (id,iselect)
    print(id,iselect,tree.name)
end
```

You will see that `iselect` is 0 for the unselection operation, and 1 for selection; id is a tree node index. These indices are always in order of appearance in a tree, starting at 0 for the root node. The `name` attribute of the tree object is the text of the currently selected node.

A pair of useful callbacks are `branchopencb` *and* `branchclosecb`. If you were displaying a potentially very large tree (like your computer's filesystem) then it would be inefficient to create the whole tree at once, especially considering that you would normally be only interested in a small part of that tree. Trapping `branchopencb` *allows you to add child nodes to your selected node before it is expanded.* `executeleafcb` is called when you double-click on a leaf, as if you were running a program in a file explorer.

In itself, the id is not particularly useful. The id order is always the same in the tree, so as nodes get added and removed, the id of a particular node will change. Generally, there is going to be some deeper data associated with a node. On a filesystem, a node represents a full path to a file or directory, or there may be an ip address associated with a computer name. IUP provides you with a way to associate arbitrary data with nodes even if the id changes. But to use this you will have to understand how to build up a tree from scratch - `TreeAddNodes` is very convenient, but won't help you if you have to add nodes later. Replace the definition of `list` and the call to `TreeAddNodes` with this code:

```
tree.name = "Animals"
tree.addbranch = "Birds"
tree.addbranch = "Crustaceans"
tree.addbranch = "Mammals"
```

You will get the top level branches of the tree; notice that they are specified in reverse order, since nodes are always added to the top. Also note the curious way in which the `addbranch` attribute is used. For a start, it is *write-only*, and the effect of setting a value to it is to add a new branch to the currently selected node. By default, this starts out as the root (which is set using the `name` attribute) The id of the root is always 0; when we add "Birds", the new branch has id 1, again when we add "Crustaceans" the new branch also has id 1 - by which time "Birds" has moved to id 2, further down the tree.

To add leaves, a similar process:

```
tree.name = "Animals"
tree.addbranch = "Mammals"
tree.addleaf1 = "Whale"
```

The `addleaf` attribute works like `addbranch`, and both of them can take an extra parameter, which is the id of the node to add to. In this case, "Whale" is a child leaf of the "Mammals" branch, which has id 1 *at this stage*. This new leaf gets an id of 2, which is one more than the parent. So this gives us a way to build up arbitrary trees, knowing the id at each point. IUP provides a function `TreeSetTableId` which can associate a Lua table with a node id. We can choose to put a string value inside this table, but it really can contain anything. Here is the first example, using some helper functions to simplify matters:

```
-- testtree2.lua

require 'iuplua'
require 'iupluacontrols'
tree = iup.tree{}

function assoc (idx,value)
    iup.TreeSetTableId(tree,idx,{value})
end

function addbranch(self,label,value)
    self.addbranch = label
    assoc(1,value or label)
```

```
end

function addleaf(self,label,value)
    self.addleaf1 = label
    assoc(2,value or label)
end

tree.name = "Animals"
addbranch(tree,"Birds")
addbranch(tree,"Crustaceans")
addleaf(tree,"Shrimp")
addleaf(tree,"Lobster")
addbranch(tree,"Mammals")
addleaf(tree,"Horse")
addleaf(tree,"Whale")

function dump (tp,id)
    local t = iup.TreeGetTable(tree,id)
    -- our string data is always the first element of this table
    print(tp,id,t and t[1])
end

function tree:branchopen_cb(id)
    dump('open',id)
end

function tree:selection_cb (id,iselect)
    if iselect == 1 then dump('select',id) end
end

f = iup.dialog{tree; title = "Tree Test"}

f:show()

iup.MainLoop()
```

There is a corresponding function `TreeGetTable` which accesses the table associated with the node id. There is also a function `TreeGetTableId` which will return the id, given the *unique* table associated with it. You can use this to programmatically select a tree node given its data by setting the `value` attribute to the returned id.

Now let's do something interesting with a tree control, a simple file browser. It is straightforward to get the files and directories contained within a directory:

```
require 'lfs'

local append = table.insert

function get_dir (path)
    local files = {}
    local dirs = {}
    for f in lfs.dir(path) do
        if f ~= '.' and f ~= '..' then
            if lfs.attributes(path..'/'..f,'mode') == 'file' then
                append(files,f)
            else
                append(dirs,f)
            end
        end
    end
    return files,dirs
end
```

We ignore '.' and '..' (the current and parent directory respectively) and check the mode to see if we have file or a directory; this requires the full path to be passed to `attributes`. This function returns two separate tables containing the *names* of the files and directories.

It is useful to define two helper functions for setting and getting data to be associated with the tree nodes:

```
tree = iup.tree {}

function set (id,value,attrib)
    iup.TreeSetTableId(tree,id,{value,attrib})
end

function get(id)
    return iup.TreeGetTable(tree,id)
end
```

Filling a tree with the contents of a directory is straightforward. We want the directories before the files, so we put them in last; nodes must be added in reverse order! The id of the new nodes will always be `id+1` where `id` is going to be the directory which we are filling. The fullpath plus a field indicating whether we are a directory is associated with each new item:

```
function fill (path,id)
    local files,dirs = get_dir(path)
    id = id + 1
    local state = "STATE"..id
    for i = #files,1,-1 do -- put the files in reverse order!
        tree.addleaf = files[i]
        set(id,path..'/'..files[i])
    end
    for i = #dirs,1,-1 do -- ditto for directories!
        tree.addbranch = dirs[i]
        set(id,path..'/'..dirs[i],'dir')
        tree[state] = "COLLAPSED"
    end
end
```

By default, the directory branches will be created in their expanded form, so we use the `STATE` attribute to force them into their collapsed state. Normally you would say this in Lua like so `state2 = "COLLAPSED"` but here we build up the appropriate attribute string with the given id and use array indexing to set the tree attribute.

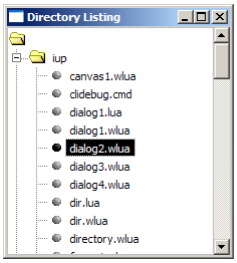
Just calling `fill('.',0)` and putting the tree into a dialog as usual will give you a directory listing of the current directory! But it would be cool if expanding a directory node would automatically fill that node; it would obviously be wasteful to fill the whole tree at startup, since your filesystem contains thousands of files. The `branchopencb` *callback is called when a user tries to expand a directory. We use this to fill the directory with its contents, but only on the _first* time that we expand this node:

```
function tree:branchopen_cb(id)
    tree.value = id
    local t = get(id)
    if t[2] == 'dir' then
        fill(t[1],id)
        set(id,t[1],'xdir')
    end
end
```

This is why directories need to be *specially marked*, so we can tell later whether we have actually generated the contents of that directory!

The first statement of this function makes the node we are opening to be the selected node of the tree. (Although we are passed the correct id of the node, it seems to be necessary to perform this step to make things work correctly.)

See `directory.lua` in the examples folder.



Menus

Any application that can perform a number of operations needs a menu. These are not difficult to create in Iuplua, although it can be a little tedious to set up. The basic idea is this: create some *items*, make a menu out of these items, and set the `menu` attribute of the dialog. The items have an associated `action` callback, which actually performs the operation.

```
-- simple-menu.lua

require( "iuplua" )

-- Creates a text, sets its value and turns on text readonly mode
text = iup.text {readonly = "YES", value = "Show or hide this text"}

item_show = iup.item {title = "Show"}
item_hide = iup.item {title = "Hide"}
item_exit = iup.item {title = "Exit"}

function item_show:action()
  text.visible = "YES"
  return iup.DEFAULT
end

function item_hide:action()
  text.visible = "NO"
  return iup.DEFAULT
end

function item_exit:action()
  return iup.CLOSE
end

menu = iup.menu {item_show,item_hide,item_exit}

-- Creates dialog with a text, sets its title and associates a menu to it
dlg = iup.dialog{text; title="Menu Example", menu=menu}

-- Shows dialog in the center of the screen
dlg:showxy(iup.CENTER,iup.CENTER)

iup.MainLoop()
```

A menu may contain items and *submenus*. This example shows a small function which makes creating arbitrarily complicated menus easier:

```
-- menu.lua

require( "iuplua" )

function default ()
    iup.Message ("Warning", "Only Exit performs an operation")
    return iup.DEFAULT
end

function do_close ()
    return iup.CLOSE
end

mmenu = {
    "File",{
        "New",default,
        "Open",default,
        "Close",default,
        "-",nil,
        "Exit",do_close,
    },
    "Edit",{
        "Copy",default,
        "Paste",default,
        "-",nil,
        "Format",{
            "DOS",default,
            "UNIX",default
        }
    }
}

function create_menu(templ)
    local items = {}
    for i = 1,#templ,2 do
        local label = templ[i]
        local data = templ[i+1]
        if type(data) == 'function' then
            item = iup.item{title = label}
            item.action = data
        elseif type(data) == 'nil' then
            item = iup.separator{}
        else
            item = iup.submenu {create_menu(data); title = label}
        end
        table.insert(items,item)
    end
    return iup.menu(items)
end

-- Creates a text, sets its value and turns on text readonly mode
```

```
text = iup.text {value = "Some text", expand = "YES"}

-- Creates dialog with a text, sets its title and associates a menu to it
dlg = iup.dialog {text; title = "Creating Menus With a Table",
      menu = create_menu(mmenu), size = "QUARTERxEIGHTH"}

-- Shows dialog in the center of the screen
dlg:showxy (iup.CENTER,iup.CENTER)

iup.MainLoop()
```

The function `create_menu` does all the work; we provide it with a Lua table containing pairs of values; the first value of a pair is always a string, and will be the label. The second value can either be a function, in which case it represents an item to be associated with a callback, or `nil`, which means that it's a separator, or otherwise must be a table, which represents a submenu. It is a nice example of how recursion can naturally handle nested structures like menus, and how Lua's flexible table definitions can make specifying such structures easy. This useful function is available in the `iupx` utility library as `iupx.menu`.

Plotting Data

Many kinds of numerical data are best seen as X-Y plots. `iup.pplot` is a control which can show several kinds of plots; you can have lines between points, show them as markers, or both together. Several series (or *datasets*) can be shown on a single plot, and a simple legend can be shown. The plot will automatically scale to view all datasets, but the default minimum and maximum x and y values can be changed. It is even possible to select points and edit them on the plot.

A simple plot is straightforward:

```
-- pplot1.lua
require( "iuplua" )
require( "iupluacontrols" )
require( "iuplua_pplot51"  )

plot = iup.pplot{TITLE = "A simple XY Plot",
                    MARGINBOTTOM="35",
                    MARGINLEFT="35",
                    AXS_XLABEL="X",
                    AXS_YLABEL="Y"
                    }

iup.PPlotBegin(plot,0)
iup.PPlotAdd(plot,0,0)
iup.PPlotAdd(plot,5,5)
iup.PPlotAdd(plot,10,7)
iup.PPlotEnd(plot)

dlg = iup.dialog{plot; title="Plot Example",size="QUARTERxQUARTER"}

dlg:show()

iup.MainLoop()
```

Creating a dataset involves calling `PPlotBegin`, a number of calls to `PPlotAdd` to add data points, and finally a call to `PPlotEnd`. You can create multiple datasets (or series) using multiple begin/end calls, and can of course use loops to add points:

```
iup.PPlotBegin(plot,0)
for x = -2,2,0.01 do
    iup.PPlotAdd(plot,x,math.sin(x))
end
iup.PPlotEnd(plot)

iup.PPlotBegin(plot,0)
for x = -2,2,0.01 do
    iup.PPlotAdd(plot,x,math.cos(x))
end
iup.PPlotEnd(plot)

plot.DS_LINEWIDTH = 3
```

A limitation of the `pplot` library is that it does not choose appropriate sizes for the plot margins. So I've had to set the bottom and left margins (in pixels) to properly accomodate the axes and their titles. As with all IupLua attributes, you can choose to make them uppercase if you like; a full list is found in the manual under Controls/Additional/IupPPlot. Some of these attributes refer to the plot as a whole, some to the *current dataset*. For instance, setting `GRID` to "YES" will draw gridlines for both axes, but if we set `DS_LINEWIDTH` to 3 after the construction of the cosine dataset, then only that line is affected.

Some attributes affect others. `DS_MODE` is used to specify how to draw the dataset; it can be "LINE", "BAR", (for a bar chart) "MARK" (just for marks) or "MARKLINE" (for lines and marks). But it has to be set before any of the other `DS_` attributes like `DS_MARKSIZE`, etc. In another case, you will often find it useful to set an explicit minimum y value by setting `AXS_YMIN`. But it will only take effect if `AXIS_YAUTOMIN` has been set to "NO" to disable auto scaling.

As with menus, making a Lua-friendly wrapper around an API is not difficult and can be very labour-saving. It would be clearer if we could work with the plot object in a more object-oriented way:

```
    plot:Begin()
    for i = 1,#xvalues do
            plot:Add(xvalues[i],yvalues[i])
    end
    plot:End()
```

And for the common case where you have arrays of values, it would be convenient to be able to say:

```
plot:AddSeries({{0,1.5},{5,4.5},{10,7.6}},{DS_MODE="MARK"})
```

Here is a function which wraps the `PPlot` API:

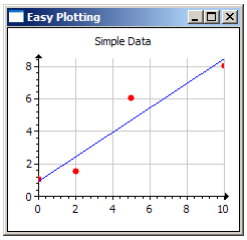
```
function create_pplot (tbl)
    -- don't need to remember this anymore!
    require( "iuplua_pplot51"  )

    -- the defaults for these values are too small, at least on my system!
    if not tbl.MARGINLEFT then tbl.MARGINLEFT = 30 end
    if not tbl.MARGINBOTTOM then tbl.MARGINBOTTOM = 35 end

    -- if we explicitly supply ranges, then auto must be switched off for that direction.
    if tbl.AXS_YMIN then tbl.AXS_YAUTOMIN = "NO" end
    if tbl.AXS_YMAX then tbl.AXS_YAUTOMAX = "NO" end
    if tbl.AXS_XMIN then tbl.AXS_XAUTOMIN = "NO" end
    if tbl.AXS_XMAX then tbl.AXS_XAUTOMAX = "NO" end

    local plot = iup.pplot(tbl)
    plot.End = iup.PPlotEnd
    plot.Add = iup.PPlotAdd
```

```
    function plot.Begin ()
        return iup.PPlotBegin(plot,0)
    end

    function plot:AddSeries(xvalues,yvalues,options)
        plot:Begin()
        -- is xvalues a table of (x,y) pairs?
        if type(xvalues[1]) == "table" then
            -- because there's only one data table, the next must be options
            options = yvalues
            for i,v in ipairs(xvalues) do
                plot:Add(v[1],v[2])
            end
        else
            for i = 1,#xvalues do
                plot:Add(xvalues[i],yvalues[i])
            end
        end
        plot:End()
        -- set any series-specific plot attributes
        if options then
            -- mode must be set before any other attributes!
            if options.DS_MODE then
                plot.DS_MODE = options.DS_MODE
                options.DS_MODE = nil
            end
            for k,v in pairs(options) do
                plot[k] = v
            end
        end
    end

    function plot:Redraw()
        plot.REDRAW='YES'
    end
    return plot
end
```

This function creates a `PPlot` object as usual, but supplies some more sensible defaults for the margins, makes setting things like `AXS_XMAX` also set `AXS_XAUTOMAX`, and adds some new methods to the object. Of these, `AddSeries` is the interesting one. It allows you to specify the data in two forms; either as two arrays of x and y values, or as a single array of x-y pairs. It also allows optionally setting `DS_` attributes, taking care to set the plot mode before any other attributes. In this way, the actual details can be hidden away from the programmer, who has then less things to worry about.

Given this function, we can write a little program which plots some points and draws the linear least-squares fit between them:

```
-- simple-pplot.lua

local xx = {0,2,5,10}
local yy = {1,1.5,6,8}

function least_squares (xx,yy)
    local xsum = 0.0
    local ysum = 0.0
    local xxsum = 0.0
    local yysum = 0.0
    local xysum = 0.0
    local n = #xx
    for i = 1,n do
        local x,y = xx[i], yy[i]
        xsum = xsum + x
        ysum = ysum + y
        xxsum = xxsum + x*x
        yysum = yysum + y*y
        xysum = xysum + x*y
    end
    local m = (xsum*ysum/n - xysum )/(xsum*xsum/n - xxsum)
    local c = (ysum - m*xsum)/n
    return m,c
end

local m,c = least_squares(xx,yy)

function eval (x) return m*x + c end

local plot = create_pplot {TITLE = "Simple Data",AXS_YMIN=0,GRID="YES"}

-- the original data
plot:AddSeries(xx,yy,{DS_MODE="MARK",DS_MARKSTYLE="CIRCLE"})

-- the least squares fit
local xmin,xmax = xx[1],xx[#xx]
plot:AddSeries({xmin,xmax},{eval(xmin),eval(xmax)})
```



`create_plot` is so useful that I've packaged it as part of the `iupx` library as `iupx.pplot`. A new pseudo-attribute has been introduced, `AXS_BOUNDS`, which is a table of four values `{xmin,ymin,xmax,ymax}`. This example shows that very different ranges can happily exist on the same plot:

```
-- pplot5.lua

require "iupx"

plot = iupx.pplot {TITLE = "Simple Data", AXS_BOUNDS={0,0,100,100}}

plot:AddSeries ({{0,0},{10,10},{20,30},{30,45}})
plot:AddSeries ({{40,40},{50,55},{60,60},{70,65}})

iupx.show_dialog{plot; title="Easy Plotting",size="QUARTERxQUARTER"}
```

<

IupLua Advanced Guide

Exchanging "Ihandle*" between C and Lua

Each binding to a version of Lua uses different features of the language in order to implement IUP handles (Ihandle) in Lua. Therefore, functions have been created to help exchange references between Lua and C.

In C, to push an Ihandle in Lua's stack, use the function:

```
void iuplua_pushihandle(lua_State *L, Ihandle *ih);
```

In C, to receive an Ihandle in a C function called from Lua, just use one of the following code:

```
Ihandle* ih = *(Ihandle**)lua_touserdata(L, pos);
```

or using parameter checking:

```
Ihandle* iuplua_checkihandle(lua_State *L, int pos);
```

In Lua, if the handle is a user data create with the above structure, but not mapped to a Lua object, use the function:

```
iup.RegisterHandle(handle, classname)
```

where "classname" is the string returned in IupGetClassName.

In Lua, to access a handle created in C as a Lua object, alternatively use the function:

```
handle = iup.GetFromC(name)
```

where "name" is the name of the element previously defined with IupSetHandle.

Error Handling

In C to improve the error report, use the following functions to execute Lua code:

```
int iuplua_dofile(lua_State *L, const char *filename);
int iuplua_dostring(lua_State *L, const char *string, const char *chunk_name);
int iuplua_dobuffer(lua_State *L, const char *string, int len, const char *chunk_name); (since 3.15)
```

These functions mimics the implementation in the standalone interpreter for Lua 5, that displays the error message followed by the stack.

If **iuplua_dofile** fail to open the given file, then it will prepend the contents of the environment variable IUPLUA_DIR to the file name and tries to open it again. (Since 3.2)

If the these functions are used, the errors will be reported through the "iup._ERRORMESSAGE(msg)" function. By default _ERRORMESSAGE is defined to show a dialog with the error message. The global attribute "LUA_ERROR_LABEL" if defined will be used in a label inside the dialog (since 3.17).

In Lua, you can also use:

```
iup.dofile(filename: string) -> (values returned by the chunk)
iup.dostring(str: string) -> (values returned by the chunk)
```

But instead of returning the error code as in C, they return the values returned by the chunk. And they will still have the same error processing as the C equivalents. (since 3.17)

OBS: When printing an Ihandle reference the returned string is "IUP(*type*): *address*", for example "IUP(dialog): 08C55240".

The Architecture of IupLua 5

There are two important names in IupLua5: "iupHandle" and "iupWidget". (renamed in 3.15)

When you create an IUP element in Lua 5 it is created a table with a metatable called "iupWidget". This metatable has its "__index" method redefined so when an index is not defined it looks for it in the "parent" table. The table it self represents the class of the control. And all the classes inherit the implementation of the base class WIDGET. Each control class must implement the "createElement" method of the class. The WIDGET class also a member called "ihandle" that contains the Ihandle* in Lua. The constructor of the WIDGET class returns the handle. The purpose of these classes is to help the creation of the smart constructors in Lua, so instead of doing "ih = iup.Label("some")" we can do "ih = iup.label{title = "some"}". It also helps to define some methods for all elements like "ih:map()", "ih:show()" and others. The BOX class inherits from WIDGET and implements the construction using elements as parameters, along with some utilities like "ih:append(child)".

The Ihandle* is represented in Lua as a table with a metatable called "iupHandle". This metable has its "__index", "__newindex" and "__eq" methods redefined. The index methods are used to implement the set and get attribute facility. The handle knows its class because it is stored in its "parent" member.

Since the controls creation is done by the "iup.<control>" function, the application does not use the WIDGET class directly. All the time the application only uses the handle.

So, for example the **IupLabel** constructor works like this:

```
iup.label calls iup.LABEL:constructor
since iup.LABEL.parent = iup.WIDGET and iup.LABEL:constructor is not implemented
it calls iup.WIDGET:constructor
then iup.WIDGET:constructor calls iup.LABEL:createElement
and finally returns the created handle
```

Custom Controls (since 3.0)

Introduction

All the IUP controls use the same internal API to implement their functionalities.

Each control, needs to export only one function that register the control so it can be used by IupCreate and other functions. Actually another utility function is exported to simplify the creation of the control.

Internally the control must implement the methods of the IUP class, and create functions that handle attributes.

See the Internal SDK for more details.

Control Class Registration

The new control must export function to register the control. This function is quite simple and it is just a call to iupRegisterClass. For example:

```
void IupXxxOpen(void)
{
  iupRegisterClass(iupXxxNewClass());
}
```

The function iupXxxNewClass is internal to the control and it creates the control class.

Control Class Implementation

The function that creates the class will (1) initialize a base class, then (2) fill its configuration parameters, (3) set the class methods, (4) register the callbacks and (5) register the attributes. For example:

```
Iclass* iupXxxNewClass(void)
{
  /* (1) - initialize the class */
  Iclass* ic = iupClassNew(NULL);

  /* (2) - configuration parameters */
  ic->name = "xxx";
  ic->format = ""; /* no creation parameters */
  ic->nativetype = IUP_TYPECONTROL;
  ic->childtype = IUP_CHILDNONE;
  ic->is_interactive = 1;
  ic->has_attrib_id = 0;

  /* (3) - class methods */
  ic->New = iupXxxGetClass;
  ic->Create = iXxxCreateMethod;
  ic->Map = iXxxMapMethod;
  ic->Destroy = iXxxDestroyMethod;
  ic->ComputeNaturalSize = iXxxComputeNaturalSizeMethod;
  ...

  /* (4) - callbacks */
  iupClassRegisterCallback(ic, "XXX_CB", "i");
  iupClassRegisterCallback(ic, "MAP_CB", "");
  iupClassRegisterCallback(ic, "HELP_CB", "");
  iupClassRegisterCallback(ic, "GETFOCUS_CB", "");
  iupClassRegisterCallback(ic, "KILLFOCUS_CB", "");
  iupClassRegisterCallback(ic, "ENTERWINDOW_CB", "");
  iupClassRegisterCallback(ic, "LEAVEWINDOW_CB", "");
  iupClassRegisterCallback(ic, "K_ANY", "i");
  ...

  /* (5) - attributes */

  /* Common */
  iupClassRegisterAttribute(ic, "SIZE", iupGetSizeAttrib, iupDlgSetSizeAttrib, NULL, IUP_NOT_MAPPED, IUP_NO_INHERIT);
  iupClassRegisterAttribute(ic, "RASTERSIZE", iupGetRasterSizeAttrib, iupDlgSetRastersizeAttrib, NULL, IUP_NOT_MAPPED, IUP_NO_INHERIT);
  iupClassRegisterAttribute(ic, "WID", iupGetWidAttrib, iupNoSetAttrib, NULL, IUP_MAPPED, IUP_NO_INHERIT);
  iupClassRegisterAttribute(ic, "FONT", NULL, iupdrvSetFontAttrib, IupGetGlobal("DEFAULTFONT"), IUP_NOT_MAPPED, IUP_NO_INHERIT);

  /* Common, but only after Map */
  iupClassRegisterAttribute(ic, "ACTIVE", iupGetActiveAttrib, iupSetActiveAttrib, "YES", IUP_MAPPED, IUP_INHERIT);
  iupClassRegisterAttribute(ic, "VISIBLE", iupGetVisibleAttrib, iupSetVisibleAttrib, "YES", IUP_MAPPED, IUP_NO_INHERIT);
  iupClassRegisterAttribute(ic, "ZORDER", NULL, iupdrvSetZorderAttrib, NULL, IUP_MAPPED, IUP_NO_INHERIT);

  /* only the default value. */
  iupClassRegisterAttribute(ic, "BORDER", NULL, NULL, "YES", IUP_NOT_MAPPED, 0);
  ...

  return ic;
}
```

You can use the iupXxxNewClass equivalent function of other controls to initialize a new base class for a new control that inherits the functionalities of the base class. For example:

```
Iclass* ic = iupClassNew(iupRegisterFindClass("canvas"));
```

You can also use the Base Class methods and attribute functions to simplify your iupXxxNewClass.

If the control is a native control then it usually will have separate modules for each driver. The iupXxxNewClass function could call a iupdrvXxxInitClass(ic) function to initialize methods and attributes that are driver dependent.

Control Creation

All controls can be created using the IupCreate functions. But it is a common practice to have a convenience function to create the control:

```
Ihandle* IupXxx(void)
{
  return IupCreate("xxx");
}
```

Control Exported Functions

The file header with the exported functions should look like this:

```
#ifndef __IUPXXX_H
#define __IUPXXX_H

#ifdef __cplusplus
extern "C" {
#endif

void IupXxxOpen(void);

Ihandle* IupXxx(void);

#ifdef __cplusplus
}
#endif

#endif
```

Tutorial

Gustavo H. S. de O. Lyrio

Antonio E. Scuri

Index

Obs: the tutorial samples are shown inside an HTML element called IFRAME. Unfortunately old browsers do not support iframes. We tested in recent versions of Mozilla **Firefox**, Google **Chrome**, Microsoft **Internet Explorer** and **Opera**. They all work ok, with some exceptions. Internet Explorer was not able to switch between C and Lua code. Firefox in Linux will try to download the contents of the iframe. If you have problems let us know.

1. Introduction

Hello and welcome to the IUP Tutorial. Our goal is to provide a walkthrough guide to develop IUP applications focused in people that haven't used IUP before. First of all it is necessary to describe what IUP is and how it can help you develop your application. IUP stands for "Interface com UsuÃ¡rio PortÃ¡til" in Portuguese, which translates to "Portable User Interface". It is a multi-platform toolkit for building graphical user interfaces, offering a simple API in two main languages C/C++ and Lua, and its purpose is to allow the user interface source code of an application to be compiled in different systems without any modification. Supported systems include: GTK+, Motif and Windows. As main advantages, IUP offers: high performance since it uses native interface

elements, and a fast learning curve due to the simplicity of its API. Also, IUP uses an abstract layout model based on the boxes-and-glue paradigm from the TeX text editor making the dialog creation task more flexible and independent from the graphics system resolution.

IUP has 3 concepts that any user has to understand: **Elements**, **Attributes** and **Callbacks**.

Elements are every kind of interface element present in the application. IUP contains several user interface elements. The library's main characteristic is the use of native elements. This means that the drawing and management of a button or text box is done by the native interface system, not by IUP. This makes the application's appearance more similar to other applications in that system. On the other hand, the application's appearance can vary from one system to another. Besides, some additional controls are drawn by IUP, and are independent from the native system. Dialogs are special elements that represent every window created by IUP. Any application that uses IUP will be composed by one or more dialogs. Every dialog can contains one or more controls inside.

Attributes are used to change or consult properties of elements. Each element has a set of attributes that affects its behavior or its appearance. Each attribute may work differently for each elements, but usually attributes with the same name work the same. Attribute names are always upper case. But attribute values like "YES", "NO", "TOP", are case insensitive, so "Yes", "no", "top", and other variations will work.

Callbacks are functions which notify the application that some user interface event occurred. Usually callbacks will be called only when the user interacts with the application elements. If the application register the callback function, then the function will be called every time the event occurs.

All we have seen until now is a short summary of what is behind the IUP toolkit and concepts that the developer needs to be familiarized with when programming with IUP. From now on, we are going to present how to build an IUP application from the most simple example possible to a complex and full of different resources application.

Previous	Index	Next

2. Hello World

2.1 Initialization

The code bellow will shows how to open an IUP environment and displays a simple message. Each line of code is explained after the code.

Example Source Code [in C] example2_1.c [in Lua] example2_1.lua

In the first line, we see an include of the C standard library, which is needed by almost all C programs. Next there is an include for the main IUP library, which is all we need for our first example. Next line is a standard main function declaration. Before running any of the IUP's functions, the function **IupOpen** must be called to initialize the toolkit. The next line creates and displays a message to the user using **IupMessage** function. This function receives from parameters: title and message. The title will be displayed at the top of the message window and the message is a text message by itself that will be displayed to the user. Following, we have a **IupClose** function call. After running the last IUP function, **IupClose** must be run so that the toolkit can free internal memory and close the interface system. Finally the program returns to exit with success.

In Lua instead of **includes** it is necessary to **require** the packages we use. The require call replace the **IupOpen** function called in C. So **IupOpen** becomes **require"iuplua"** and like the includes it is done at the top of the code. This will inform Lua that the program described in the next lines makes use of the package iuplua. Other important change from C to Lua is the way that IUP functions are called. The IUP calls in Lua uses the prefix "iup." instead of "Iup" informing Lua to search for the function inside iuplua package. So, in this example we have: iup.Message instead of **IupMessage** and iup.Close instead of **IupClose**. Another important aspect of using IUP with Lua is the fact that Lua is a interpreted language and a Lua program execute from the first line to the last one. So, there is no need to create a main function. We just call the iup.Message after the package inclusion. Yet, in the Lua version of our example you will find a call to **iup.MainLoopLevel**. It will check if **iup.MainLoop** was already called to avoid calling it again. This is useful only if your script could be executed from inside another context, for regular applications there is no need for calling it.

2.1.1 Compiling and Linking

Compiling and Linking a program that uses IUP (as any other third party library that is not installed on the system) demands that you specify where the **include** files and the **libraries** are installed. You also need to link with the iup library. In order to do that in a single command line for our first example is as follows:

```
gcc -I/tecgraf/iup/include -L/tecgraf/iup/lib -liup -o example2_1 example2_1.c
```

For programs containing several modules we suggest building a makefile (See here how to build one: makefile tutorial). There are also many different IDEs (Integrated Development Environments) in Linux and in Windows that can help you develop an application. They all need the same basic settings to be configured. We also have guides available for the most popular IDEs:

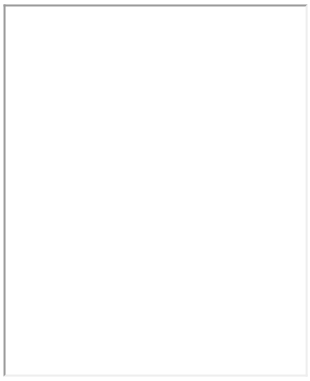
- Borland C++ BuilderX
- Code Blocks
- Dev-C++
- Eclipse for C++
- Microsoft Visual C++ (Visual Studio 2003)
- Microsoft Visual C++ (Visual Studio 2005)
- Open Watcom

If you want more details on libraries dependencies for static linking, you can check the Building Applications Guide.

2.2 Creating a Dialog

Let's change the first example a little bit to add our own dialog.

Example Source Code [in C] example2_2.c [in Lua] example2_2.lua



Note that we have added a new line in which we declare Ihandles* variables for IUP elements. We also have created two different variables. One called **dlg** for our main dialog and another called **label**, which will hold a label with a hello message. Next, a new line creates a iup label control and associates it with the label Ihandle that was previously declared. Its only argument is the text that will be displayed inside the label. Then we reach the line in which we create our main dialog, almost in the same way that we created the button. The difference goes on the parameter passed to **IupDialog** function. It receives another function that will create a composition control called **IupVbox**. A **IupVbox** is a control that aligns all controls passed to it vertically. In this example, we are passing just one control (our label) and a NULL to sign that we are done with our list of elements. Next line presents the way in which IUP changes each control attributes. By calling the function **IupSetAttribute** the programmer will inform which control has the attribute that needs to be changed, which attribute is that and the new value that the attribute will assume. In our sample, we are changing the main dialog's title to "Hello from IUP Tutorial". The next function is called **IupShowXY** and tells IUP that we need the main dialog displayed at the center of the screen horizontally and vertically. Following comes one of the most important function which is called in our program: **IupMainLoop**. This function tells iup to wait for events. Otherwise, the program would go on, end and terminate without dealing with any event. Go on, comment this line, recompile your code and execute your program, and you will see the main dialog blink in the screen and the program ends just after it. It will be a valuable exercise.

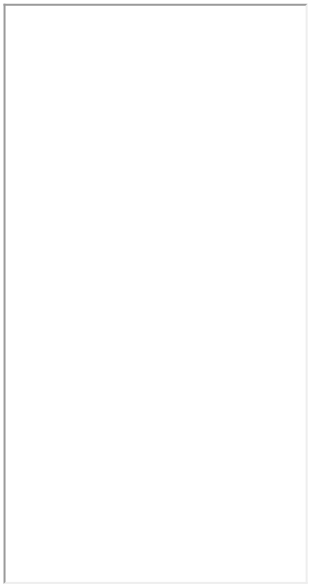
From the most simple hello world to the most complex IUP application, all will have this same code structure.

In Lua we created the controls just after the iuplua package require. The controls construction have a different form in Lua, where the constructor name is all lower and its parameters are inside a table. Although you can, there is no need to use **iup.SetAttribute**. In Lua a control is also a table where the fields are the control attributes and callbacks. In the example all attributes were defined during the control creation, but we could also do label.title or dlg.title after calling the constructor. In Lua some functions also have a syntax sugar, so instead of using "iup.ShowXY(dlg, iup.CENTER,iup.CENTER)" we can write "dlg::showxy(iup.CENTER,iup.CENTER)" but both are exactly the same call.

2.3 Adding Interaction

In the previous section, we saw how to build a basic IUP application, but without any custom interaction with the dialog. In this section, we will add interaction to our application using a button.

Example Source Code [in C] example2_3.c [in Lua] example2_3.lua



After the usual includes, we find some new lines. These lines contain a regular function called btn_exit_cb that will be registered as our button callback, as will be seen next. This function does nothing special, except showing the hello message that we saw in the first example and also closing the application returning code IUP_CLOSE.

Note that we have added a new handle that will handle our vbox in a clear way. Following is our button declaration. The first parameter is the title for the label, and the second parameter is a global name for a callback which use is now deprecated, so we simply set to NULL. The next lines are our vbox, which now is using a variable. That variable is passed as a parameter to the **IupDialog** function.

As said before, callbacks are special functions defined by the programmer and called by IUP when an event needs to be handled. To create a callback, the programmer must declare a function and put inside its body anything that he/she wants the application to do when the event occurs. After that, it is necessary to inform IUP that new function is, in fact, a callback. That is done calling the function **IupSetCallback**. This call will inform IUP that our regular function btn_exit_cb is actually a callback that needs to be executed when our button is pressed. The first parameter is our button Ihandle, followed by the name of the callback and the name of the function to be called, casted as Icallback. The names of the available callbacks can be found at each control documentation. As attribute names, they are always written in upper case letters.

When executed, the application's dialog box will show up, and when the user presses the button, it displays a hello message and will close the application. It seems not a big deal, but with this small sample of code, we have covered the process of creating an IUP application, declare elements and callbacks, and also handle an event. From now on, we are going to see more from IUP controls and how to improve our application using different kinds of controls.

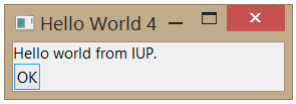
In Lua we set the button action callback just like we set an attribute. So, there is no call to **IupSetCallback**.

2.4 Adding Layout Elements

Up until now we have just positioned our controls inside a vbox which, as told, aligns all controls inside it vertically. This is just a small sample of the IUP's layout concept. IUP implements an abstract layout, in which the positioning of controls is done relatively instead of absolutely. For such, composition elements are necessary for composing the interface elements. They are boxes and fills invisible to the user, but they play an important part. When a dialog size changes, these containers expand or retract to adjust the positioning of the controls to the new situation allowing the dialog to adapt even if the resolution of the screen changes. That would come in hand if you port your application to another system with a lower resolution, for example. Main composition elements are vertical boxes (vbox), horizontal boxes (hbox) and filling (fill), among others. There is also a depth box (zbox), in which layers of elements can be created for the same dialog, and the elements in each layer are only visible when that given layer is active.

To clarify the way abstract layout works, lets modify our example adding a label to it.

Example Source Code [in C] example2_4.c [in Lua] example2_4.lua



Note that there is a new **label** declaration and this new element appears inside our vbox as the top element. That means it will be displayed above **button**, and that's all. Our example now has two different elements and is disposed vertically one above the other. An interesting exercise would be to change the code above and use an hbox to see what happens.

In Lua we used a simpler way to associate a callback, using a Lua syntax sugar "button:exit_cb()". But to use this syntax, the button element must exist before the callback declaration, so we also moved its code to after the button construction.

2.5 Improving the Layout

Now that you understand the basics of abstract layout, let us present three attributes available to both vboxes and hboxes. They are: ALIGNMENT, GAP and MARGIN.

ALIGNMENT defines the horizontal or vertical alignment of elements inside the box. If you are using a vbox, it will be an horizontal alignment, or if you are using an hbox, it will be a vertical alignment. Its values can be "ALEFT", "ACENTER" or "ARIGHT" for horizontal alignment, and "ATOP", "ACENTER" or "ABOTTOM" for vertical alignment. The default value is "ALEFT" and "ATOP".

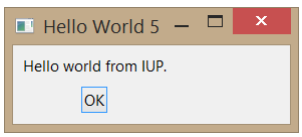
GAP defines a space in pixels between every element inside the box. If you are using a vbox, it will be a vertical space, or if you are using a hbox, it will be a horizontal space. The default value for GAP is 0 (which means no space between elements).

MARGIN defines a margin in pixels. Its value has the format "widthxheight", in which width and height are integer values corresponding to the horizontal and vertical margins, respectively. Its default value is "0x0" (means no margin).

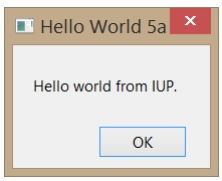
Let's see how our layout responds to these three attributes.

Notice that the Lua version of our example has only a few changes. We just add a line with the label creation and inserted it into the vbox a little further.

Example Source Code [in C] example2_5.c [in Lua] example2_5.lua



After creating the vbox, we have added three lines that set those attributes to values different than the default values. The result is much pleasanter to see. Although it is still not quite as the first example, which uses a pre-defined dialog. Can you figure out which attributes we need to set in order to obtain a more closer appearance?



Previous | Index | Next

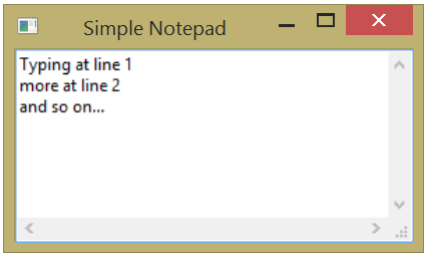
3. Simple Notepad

3.1 Main Dialog

Until now we have seen two different controls: labels and buttons. Labels can show text or images to the user but are not designed for interaction. Buttons allow the user to trigger an event by pressing a mouse button. But none allow the user to insert any data into our application. To do that,we will use a new control called **IupText**. It creates an editable text field and has a lot of different attributes available. We will be interested in one in particular for now: MULTILINE. MULTILINE turns the **IupText** into an editable text field that supports many lines, which is mandatory to build a simple notepad.

Our starting code for the simple notepad should be as follows.

Example Source Code [in C] example3_1.c [in Lua] example3_1.lua



The previous code doesn't show exciting news except by the **IupText** declaration and the call to **IupSetAttribute** to set the **IupText** as a MULTILINE. The default value is "NO", try to comment this line and see what happens.

Notice that the SIZE attribute of the dialog was also set. Since the **IupText** is a control that does not fit its size to its contents, we have to set an initial size for the dialog, or else the result would be a very small dialog. We use a simple size specification that is a quarter of the screen size in both dimensions. The SIZE attribute will also work as a minimum size, so we reset the USERSIZE attribute, after the dialog is shown, to avoid this limitation. Try to comment this line and check out how the dialog interactive resize behaves.

With a few lines of code, we build an application where the user can type a huge text. But, if you type a huge text, you probably would like to save it, and unfortunately our applications offers no such feature. We will handle this in the next sections.

3.2 Adding a Menu

Almost all applications offer a menu where the user can load files, save files, use the clipboard and do a lot of other stuff with his data. IUP also offers this resource to the applications. Menus are divided into four different interface elements: **IupItem**, **IupMenu**, **IupSeparator**, **IupSubmenu**.

IupItem creates a single item of the menu interface element. When selected, it generates an action.

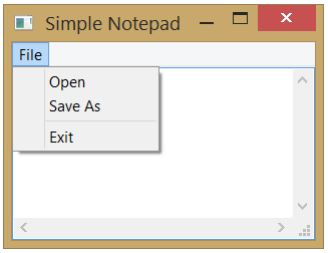
IupSeparator creates a horizontal line that will appear between two menu items. It is normally used to divide and arrange different groups of menu items.

IupSubmenu creates an item that, when selected, opens another menu.

IupMenu creates the menu element by itself as a list of elements. An **IupMenu** can include any number of the other 3 types of menu interface elements: **IupItem**, **IupSubmenu** and **IupSeparator**. Any other type of interface element inserted in a menu will be an error.

Let's add a menu with a few items in our example.

Example Source Code [in C] example3_2.c [in Lua] example3_2.lua



Now our example has a few menu element handlers and declarations. Also, we used our exit callback to be called when the item_exit menu item is selected. The next line shows the composition of an **IupMenu** called file_menu. Note that the menu items are passed in order of appearance, which means that item_open will appear above item_save and so on. There is also an **IupSeparator** dividing our file menu in two parts, the first takes items that deal direct with files, like open and save, and the second takes the exit item. It's not mandatory to have an **IupSeparator** in your menu. This is used just to keep things more organized. Next line is a little tricky. We created a submenu to store all of our items. Why not use file_menu directly? We could, but it would be used as main menu and would end up being the only menu available in our application. It's a good practice to separate menus in submenus and then pass these submenus as items of the main menu. By doing so, an application could have a file menu, a search menu, a help menu, and others as items of the main menu, as you can see in the main menu declaration on the next line.

At last, once we are done building the main menu, we must set the MENU attribute of the main dialog as the menu we have just created. But since it is neither a string nor a number, we must use a different function to do this association, which is called **IupSetAttributeHandle**.

You should notice that the exit menu item works fine, as we set the *Exit* menu item action callback, but *Open* and *Save* still don't work. That's because we didn't set any callback for them. Those callbacks will use another IUP feature, which is the subject of our next section.

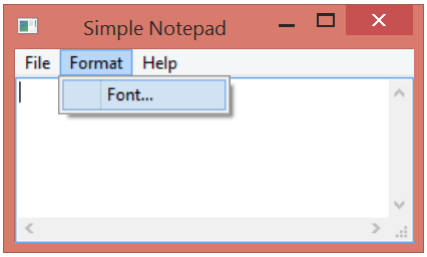
3.3 Using Pre-defined Dialogs

In the previous section, we added a file open and a file save menu items, but they had no callbacks associated. That's because we will use new IUP resources to deal with file handling. These resources are called Pre-defined Dialogs.

Some dialogs are commonly found in a lot of different applications like file selection dialogs, font selection dialogs, color selection dialogs, etc. It would be annoying to have to build the same dialog again every time we need to select a file, or to select a color or a font. So, IUP provides pre-defined dialogs with all the necessary controls to deal with these common tasks.

We will update our last example to handle file input/output and to make use of these IUP pre-defined dialogs.

Example Source Code [in C] example3_3.c [in Lua] example3_3.lua



We will need to access the multitext control from inside the menu callbacks. There are many ways to do that; the simplest one is to declare it as a global variable. We will do that to illustrate this example, but this is not recommended. In the next example, we will show you how to not use a global variable to obtain the same results.

Now we have interesting new functions. First, let's take a look at the new callback called open_cb. This callback will handle the file opening when the user clicks on the Open menu item. For this we will use a IUP predefined dialog called **IupFileDlg**. This dialog is a standard file- handling dialog with all the features that we need to select a file from the file system, and it will also save a lot of work. Inside the callback we create our **IupFileDlg**, and set it to be an "open" dialog with attribute DIALOGTYPE. Also we set EXTFILTER attribute to "Text Files|*.txt|All Files|*.*|", since we want our application to handle text files but we leave the option for listing other files.

Now the program calls **IupPopup**, which is a function similar to **IupShow**, but it restricts the user interaction only in the specified dialog. It is the equivalent of creating a Modal dialog in some toolkits. Its arguments are our file dialog Ihandle followed by x and y coordinates that we defined as the center of the screen with IUP_CENTER.

Then we have a conditional test in which we get the value of filedlg STATUS with **IupGetInt**. Why not use **IupGetAttribute** instead? That's because **IupGetAttribute** returns attributes as strings, but we know that STATUS is an integer so we can simplify our status check using **IupGetInt**.

Once our file dialog returns a valid status, we are able to recover the name of the selected file using **IupGetAttribute** to retrieve the VALUE attribute. Then we read the file using a simple function and fill in its contents on the multitext control by using the **IupSetStrAttribute** function to set its VALUE attribute. We can not use the **IupSetAttribute** function, because our C string returned by **IupGetAttribute** is a dynamically allocated pointer. Therefore **IupSetStrAttribute** will make sure that the string is duplicated internally and not dependent on the given pointer.

Now we are done with this dialog. You can simply call **IupDestroy** to remove filedlg from memory, because we will not need it anymore.

Next there is another callback, saveas_cb, which will select a file name for saving the content of a file. It is very similar to open_cb, but DIALOGTYPE is set to SAVE, so this time it will select a file name for saving. In this case the filename can be also for a new file, if an existing file then the user will be notified of overwriting so it can cancel and start over. After selecting the filename we are going to save the multitext contents to the file.

Now comes the font_cb callback that, as you may have already guessed, will call a predefined dialog to select a font. To do that, we use **IupFontDlg** instead of **IupFileDlg**. To set the font, just change the FONT attribute in the multitext control.

The next callback is about_cb, which does nothing special, just calls **IupMessage** to display a text to the user.

The following lines don't show anything new, except for the new callbacks registration. But notice that we added "..." to the text of the menu items in which a dialog is open. This is not mandatory, but is highly recommended by common User Interface Guidelines.

Finally, we now have a brand new text editor using IUP. But what happens if the dialog that your application needs is not provided by IUP as a predefined dialog? That will be the subject of our next section.

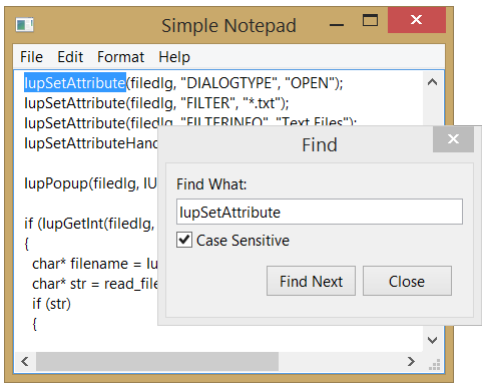
3.4 Custom Dialogs

We saw in the previous section that IUP provides predefined dialogs that can be used by the applications to save a lot of developing time. But if the dialog your application needs is not one of IUP's predefined dialogs, then it's time to built your own dialog. The good news is that you have already made this when building your main dialog. The tricky part here is how to handle more than one dialog at the same time.

For this we will add two new items to our Edit menu: Find and Go To. Find will search the multitext contents while looking for a string and highlight it when found. It will search for this string

many times, and the search can also be case sensitive. Go To will position the caret to a specific line in the text.

Example Source Code [in C] example3_4.c [in Lua] example3_4.lua



The first change is the inclusion of two utility functions (str_compare and str_find) that will be used to implement Find, and which are not the object of this tutorial. If you want to understand what is inside these functions, take a closer look into the code.

You will notice that several functions had their names changed from the previous example code. We did that to illustrate the importance of function nomenclature in a larger project, so that several callbacks can be easily associated with their respective control. For instance, open_cb became item_open_action_cb, saveas_cb became item_saveas_action_cb, and so on.

Allow me to make a jump in our code and please refer now to the item_find_action_cb function. This callback, despite being almost at the end of the code, is responsible for building one of our custom dialogs. In this dialog, we will use some elements that we have already seen in previous sections: a text field to receive the string that the user wants to find, a button to find the next occurrence of this string, a button to close our find dialog, and two new IUP elements: **IupToggle** and **IupFill**.

IupToggle is a two-state (on/off) button that, when selected, execute a callback. Toggles are normally used to set flags. In this case, we used it to allow the user to decide if the search will be case sensitive or not.

IupFill is a very peculiar element. It is, as the name says, used to fill blank spaces inside our dialog. In other words, it positions and aligns IUP elements. The best way to understand IupFill is to think of it as a coil spring. If you put an **IupFill** inside an **IupHbox**, it will expand between the two elements, pushing one to the left and the other to the right. Or if you put it in an **IupVbox**, above a element, it will push the element all the way down. But IupFill also has a SIZE attribute, that can be used to control how much space will be taken. With experience we will find the correct way to define SIZES for **IupFill** and for other elements as well. In our case, **IupFill** is being used to push the buttons Find Next and Close to the right, inside our hbox.

Note that our new dialog has a lot of new parameters set. DIALOGFRAME will remove minbox, maxbox, and it will resize from the corner of the dialog. This will provide a reduced functionality and a standard dialog box appearance. DEFAULTENTER defines a button to be activated when the users presses ENTER, in this case it will have the same effect as pressing the next_bt button. DEFAULTESC works the same way for the ESC key by activating the close_bt button. Next the attribute PARENTALDIALOG sets the dialog that holds item_find (our main dialog) as the parent of our new dialog, by using **IupGetDialog**, which returns the handle of the dialog that contains the element passed as parameter. This will maintain the Find dialog always on top of the main dialog, even if we change the focus to the main dialog. It will also allow us to set the find dialog position at the center of the parent dialog.

In the next two lines, we use custom attributes to store application pointers. Each IUP element can hold as many custom attributes as you want. If your application needs to store some information to be retrieved later, you can just set it as we are doing here. We created a new attribute called MULTITEXT in the dialog to store the multitext element pointer and make it available to other callbacks. Doing this, we avoid the global attribute used in the previous example. Also, we created another new attribute called FIND_DIALOG in the element find_item, so we will be able to reuse this dialog. Everytime this function is called, the dialog is not created again, since it is created only once.

Next we show our dialog using **IupShowXY** and pass IUP_CURRENT to it. At first, this will center the dialog according to its parent (main dialog as we defined above). Next time it will reuse the last position, since the dialog will not be destroyed when closed.

Now that we have built the Find dialog, it is time to write the callbacks that will effectively do the job to find the string inside our multitext.

Let's turn our attention to the find_next_action_cb callback. This callback is responsible for finding the next occurrence of our string inside the multitext and it has a lot new function calls. We call to **IupGetDialogChild**, which is a function that returns the identifier of the child element that has the NAME attribute in the same dialog hierarchy. We use this to retrieve the multitext handle. This is a more elegant form to retrieve handles, instead of using a custom attribute or making a global variable, but it only works for the same dialog. Next we retrieve the text to be found and the case sensitive flag from the respective controls. The search is performed, and if the result is positive, we will save the last found position in a custom attribute, and call **IupSetFocus**. When we showed our Find dialog, we moved the focus from our multitext to the new dialog. This function restors the focus to the multitext. We then select the text on the multitext. Next we find two calls to **IupTextConvertPosToLinCol** and **IupTextConvertLinColToPos**. These are used to compute the position we use to scroll the multitext, so the selection becomes visible.

Beside next_bt, find dialog also has close_bt, and it also demands a callback. Find_close_action_cb closes the Find dialog. In this callback, we made a call to **IupHide**. When a dialog is hidden, it is not destroyed, so you can show it again.

The Go To dialog will work in the same way. If you have understood how to create the Find dialog, you should be able to build the Go To dialogs.

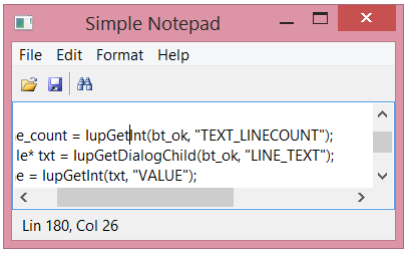
3.5 Adding a Toolbar and a Statusbar

Now that we saw how to use predefined dialogs or how to build our own dialogs, lets see how to implement two other resources present in many other applications: toolbars and statusbars.

Toolbars are a set of buttons, usually positioned side by side in the top of the dialog, just bellow the menu. To build our toolbar, we will use the attribute IMAGE of **IupButton**. As in predefined dialogs, IUP also offers a series of predefined images to be used with buttons. These images are part of an additional library called **IupImageLib**. To use this library, call **IupImageLibOpen** right after **IupOpen**.

Statusbars normally appear on the bottom of the dialog and usually show some information about what is happening inside the application. To build our statusbar, we will use a set of **IupLabel** controls arranged side by side. In our statusbar, we will be displaying the caret position in the text, and to achieve this, we will use a IupText callback called CARET_CB, which is called every time the caret position is changed.

Example Source Code [in C] example3_5.c [in Lua] example3_5.lua



The first change, as told above, is the inclusion of multitext_caret_cb to our callbacks. In this callback, we will make use of the parameter received by the callback. First we retrieve the handle of **IupLabel** called lbl_statusbar using **IupGetDialogChild**, and then passing the handle that came as a parameter of our callback. Next, we set the label's TITLE by building a string using lin and col parameters, in which the callback provides the caret line and column position.

From this new callback, we will jump to main function where the next change appears. Just after IupOpen, you will find a call to **IupImageLibOpen**. This function will load the image library, so we can use its images in our toolbar.

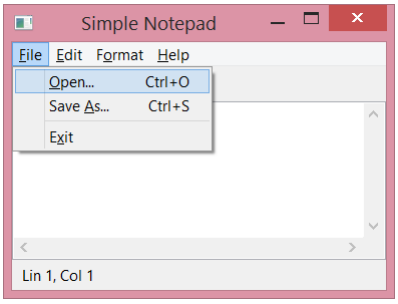
A few lines after, we will find our lbl_statusbar declaration. This label will play the role of our statusbar. It needs the EXPAND attribute set to HORIZONTAL, so it will occupy all the horizontal space inside the vbox. Following we will see some button declarations (btn_open, btn_save and btn_find) and some calls to **IupSetAttribute** setting each button's image. The images names can be found at the **IupImageLib** documentation. We then notice that our toolbar is nothing more than an **IupHbox** containing those buttons. Note, a few lines after, that we set the buttons callbacks to the same callbacks set for the respective menu items. This is feasible because the buttons do exactly the same thing as the items representing a short cut to call open, save or find. We also set the FLAT attribute for the buttons so their border is removed, and they will look like toolbar buttons. We set the CANFOCUS attribute to No for the buttons so they will not receive the keyboard focus as toolbar buttons behave.

The final change will be the inclusion of toolbar_hb and lbl_statusbar in the vbox that already had our multitext. The toolbar comes first because it is a vbox and we want it above the multitext. lbl_statusbar goes after because we want it bellow the multitext. That's all. Our application now has both a toolbar and a statusbar. In the next section, we will improve it even more by adding hot keys to our menus.

3.6 Defining Hot Keys

Applications that have menus always present hotkeys to its users. IUP also offers this resource. To define a hotkey, you could use **IupDialog** callback **K_ANY**. This is a callback common to a lot of IUP elements and is called when a keyboard event occur. IUP also offers a simple way that allows you to define a specific callback for the key combination you want to deal with. For example, if you want to show the file Open selection dialog when the user presses Ctrl+O, you just have to set a callback called "K_cO". **Keyboard Codes** shows a complete table with all keyboard codes available in IUP.

Example Source Code [in C] example3_6.c [in Lua] example3_6.lua



This example didn't change much. We just added "\tCtr+?" to each menu item that has a hotkey. Character "\t" will take care of aligning our hotkey text to the right, and the rest of the string will tell the user which key combinations to press. Note that "?" should be replaced by the key you want.

A few lines after, we did a few calls to IupSetCallback using key combinations as callback names, as mentioned above, to deal with key pressed events. That's all we need to change to include hotkeys in our application.

Since we are improving the user keyboard experience, there is another feature that we can use to aid users. Using the ampersand (&) character in the menu item text, we define a key that can activate the menu item. The next character following the ampersand will be the key. The main menu is reached using the Alt+key combination, for instance Alt+F will activate the File menu. Once the menu is opened, use the 'O' key to activate the file Open menu item. Another example is the Alt+F then 'X' key combination to exit the application; many applications have this key combination enabled.

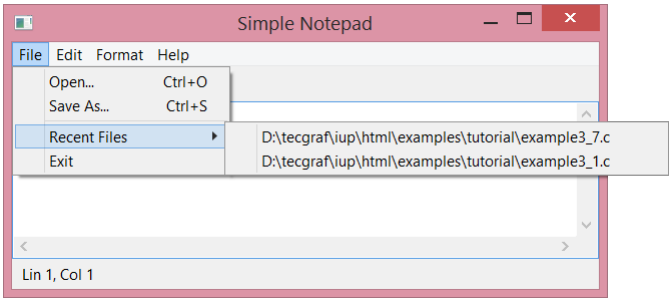
Finally we add the TIP attribute for the toolbar buttons so they will also show the key combination that activate its feature.

3.7 Recent Files Menu and a Configuration File

Many text editors offer a menu item that holds a list of recent files. We will use an IUP resource called **IupConfig** to implement this list and also store other configuration variables. IupConfig implements a group of functions to load, store and save application configuration variables. For example: the list of Recent Files, the last position and size of a dialog, last used parameters in dialogs, etc. Each variable has a key name, a value and a group that it belongs to.

Its important to remember that using **IupConfig** demands the inclusion of header file iup_config.h.

Example Source Code [in C] example3_7.c [in Lua] example3_7.lua



Note that in this new example we have included the iup_config.h header as advised above. We will start this analysis from our main function. After creating a handle for our config by calling **IupConfig**, we set the attribute APP_NAME. This attribute defines the name of our configuration file. In UNIX, the filename will be "<HOME>/.<APP_NAME>", where "<HOME>" is replaced by the "HOME" environment variable contents, and <APP_NAME> replaced by the APP_NAME attribute value. In Windows, the filename will be "<HOMEDRIVE><HOMEPATH>\<APP_NAME>.cfg", in which HOMEDRIVE and HOMEPATH are also obtained from environment variables.

After that comes a call to **IupConfigLoad** that will load our config file at startup. This function combined with the **IupConfigSave** function, which we will see later, will allow our configuration variables to be persistent between different application executions.

Following, a few lines bellow, we create the recent_menu that will hold our recent items inside. You will see that it works as any other menu creation, except by the fact that we will not add any menu items. They will be provided by a function that we will see soon. We positioned our recent_menu above the item_exit menu item and bellow the **IupSeparator**.

After the menu is created, there is call to **IupConfigRecentInit**. This function is responsible for initializing the recent_menu items from the configuration file entries. The item_recent_cb callback will be called when the user selects a file in the recent list. This function also defines the number of recent files that will be stored and displayed. In our example, we choose to store 10 files. Also note that both item_open_cb and item_saveas_cb should change the recent files list. So a call to **IupConfigRecentUpdate** is necessary to maintain our recent files list updated.

Next line shows a call to **IupConfigDialogShow** that replaces **IupShow/IupShowXY**. This function will also show the dialog, but it will try to use the last position and size stored in the configuration file. It can be used for any application dialog, just use different names for each dialog.

The function **IupConfigDialogClosed** is used to save the last dialog position and size when the dialog is about to be closed, usually inside the dialog CLOSE_CB callback, or when the dialog is programmatically hidden. The CLOSE_CB callback is called when the user clicks on the dialog close button, usually a 'X' at the top right corner of the dialog. Here our dialog is closed by the item_exit_action_cb callback, so we decided to also use this function as the CLOSE_CB callback and to call **IupConfigDialogClosed**. Finally, since this callback also exists in the application we use to call **IupConfigSave**, it will save our configuration file. Now that it is saved, we can destroy its handle using **IupDestroy**.

That's all for the main function, so let's turn our attention to the item_recent_cb callback. This callback, as said before, is responsible for handling the selection of a recent file at the menu_recent. Inside it, we recover our config handle from a custom attribute in the dialog, then we get the file name from the TITLE attribute of item_recent and open it the same way we do in item_open_cb.

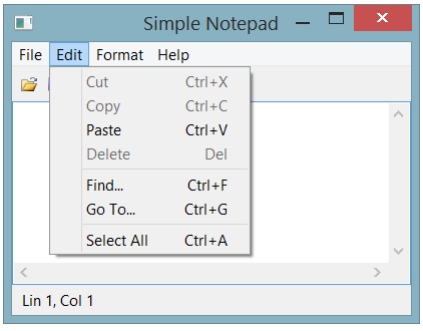
3.8 Clipboard Support

Next, we will find some callbacks that handle copy, cut, paste, delete and select all of the new items added to the Edit menu, and a callback to manage activation and deactivation of these items. All are very short callbacks.

item_copy_action_cb, item_paste_action_cb and item_cut_action_cb use a resource called **IupClipboard**, which creates an element that allows access to the clipboard. Each **IupClipboard** should be destroyed using **IupDestroy**. You can use only one for the entire application, because it does not store any data inside, or you can simply create and destroy every time you need to copy or paste, that's how we did in our notepad. The item_copy_action_cb callback retrieves the SELECTEDTEXT attribute from our multitext and sets the clipboard TEXT attribute to copy the text selection. item_paste_action_cb retrieves the clipboard TEXT attribute and insert it (paste) in the multitext, where the cursor is positioned, using the INSERT attribute. item_cut_action_cb is almost the same code as copy, except by the fact that it sets attribute SELECTEDTEXT to "", removing the selected text from the multitext. item_delete_action_cb does the same as cut, but without using the clipboard. item_select_all_cb sets the attribute SELECTION to ALL, selecting all the text inside the multitext.

Another callback was created to deal with the initialization of our new menu items. edit_menu_open_cb is associated to the edit_menu OPEN_CB callback. It will set the cut, paste, copy, and delete items as active or inactive, depending on some conditions. First, it is necessary to obtain the handles of these items. We use the NAME attribute of each item and the **IupGetDialogChild** function for this propose, just like we did before for the multitext. We then test if there is text available in the clipboard by calling **IupGetInt**(clipboard, "TEXTAVAILABLE"). It is a short way to test a boolean return value, without having to compare strings with "YES" or "NO". So, if it returns 0, it means there is no text in the clipboard, or in other words, there is nothing to paste. Then the Item paste should be disable, by setting ACTIVE to "NO". Otherwise, the user should be able to paste, and we should set ACTIVE to "YES". The other items follow the same idea, but this time checking the content of the attribute SELECTEDTEXT. If there is nothing selected, you can disable cut, copy and delete items. Otherwise, you can enable all items.

Example Source Code [in C] example3_8.c [in Lua] example3_8.lua



3.9 More File Management (Drag&Drop, Command Line, ...)

In this section, we will see a little more of file management. The example will show you how to handle drag and drop support, command line support, and how to check if the file needs to be saved before taking another action.

First we will find some new auxiliary functions called **str_filetitle**, **new_file**, **open_file**, **save_file**, **saveas_file** and **save_check**.

str_filetitle will be used to append the name of the file opened by the application to the application dialog title. **new_file** first retrieves the main dialog and the multitext, then sets the dialog title to "Untitled - Simple Notepad" and multitext attributes FILENAME to NULL, VALUE to "", and the new attribute DIRTY to "NO". DIRTY is a custom attribute that we created (same way we did with FILENAME) to check if the multitext has changed and has not been saved. Every time the multitext text is changed, the callback VALUECHANGED_CB, named multitext_valuechanged_cb, is called to set DIRTY as "YES". This attribute will allow us to identify if the content of the multitext has changed and needs to be saved. **open_file** reads the file and sets almost the same attributes as **new_file**, except by the fact that it uses **str_filetitle** to set application title with the filename, and it also sets the file content into multitext VALUE attribute. Notice that, like **new_file**, it also sets DIRTY to "NO". Since we have just opened the file, it doesn't need to be saved. Also, **open_file** calls **IupConfigRecentUpdate** to include the file we just opened in the recent files list. **save_file** calls write_file to save the current file and sets the DIRTY attribute to "NO" while **saveas_file** does the same but replacing the current opened file for the new edited one updating the

recent list with the new file. Finally, **save_check** uses the DIRTY attribute to check if the file needs to be saved. We used **IupGetInt** to automatically convert DIRTY from "YES" or "NO" to 1 or 0. If it's 1, we then call a predefined dialog called **IupAlarm** to warn the user that the multitext content has changed, and it is not saved. If the user chooses button 1 - "YES", it will call our item_save_action_cb to save the file. If the user chooses button 2 ("No") **save_check** will returns 1 without saving and continue with the application operation, but if user chooses button 3 ("Cancel") **save_check** will return 0 meaning that no further action should be taken.

We then reach the callbacks section. This time we added several new callbacks. First is **dropfiles_cb** that will handle what happens when a file is dropped inside the application. It is a very simple callback, it simply checks if the current multitext needs to be saved with **save_check**, and it calls **open_file** to open the file that came as a parameter. What is important here is to notice that we associated this callback with two different DROPFILE_CB. One for the multitext and one for the main dialog. The reason is that the user may drop the file in any place inside the dialog. Every IUP control will call the main dialog DROPFILE_CB , except by the multiline. So, if we want the file to be opened, when it is dropped inside the multiline, it's necessary to set the multitext DROPFILE_CB callback as well.

The next one is a multitext_valuechanged_cb that, as we mentioned before, is called when the user changes the text inside multiline. It simply sets the DIRTY attribute as "YES". It is set as the multitext VALUECHANGED_CB callback in main function.

Next comes file_menu_open_cb, which is a function called when the user clicks on the file menu, but before it is displayed to the user. We will use it to enable or disable the save and the revert items, depending on the DIRTY attribute value.

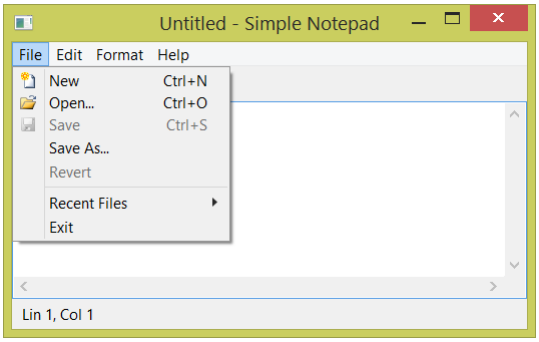
We have also changed config_recent_cb, that now checks if the current multiline content needs to be saved and calls our new function **open_file**. Another new callback is item_new_action_cb that does the same, but calls **new_file** instead. item_open_action_cb has also been changed to use **save_check** and **open_file**, as well as item_exit_action_cb.

Two more callbacks were created: item_save_action_cb and item_revert_action_cb. item_save_action_cb saves the text in a file using the current filename we stored in the FILENAME attribute, without displaying the save as dialog. If there is no current filename, it calls item_saveas_action_cb, so the user can choose a file. item_revert_action_cb reopens the current file, discarding any changes made to the multiline content.

In the main function, we have new menu items: item_new, item_save, and item_revert. Also we have new buttons: btn_cut, *btn_copy, i *btn_paste, and *btn_new. Multiline has the new DIRTY attribute and two new callbacks: VALUECHANGED_CB and DROPFILES_CB. And we made a call to **new_file** to initialize the application state, as a new file has just been created. Also the callbacks where rearranged to appear next to its items to made the code more clear. Finally, in the next line, we use argc and argv to check if the user tried to open a file from the command line, if so, we call **open_file**.

That's all for file handling. Let's proceed to the next section, where we will work with IUP dynamic layout.

Example Source Code [in C] example3_9.c [in Lua] example3_9.lua



3.10 Dynamic Layout

Now we would like to be able to hide and show some of the complementary dialog elements, such as the Toolbar and the Statusbar. If you simply set their VISIBLE attribute to NO, they will be hidden, but their size in the dialog layout will still be reserved, and an empty space will be displayed instead. To avoid the use of the FLOATING attribute, along with the VISIBLE attribute, they will be hidden, and its space will be used by the multitext.

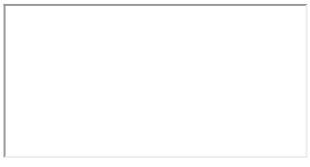
To implement this feature, we added a new submenu called "View" at the main menu, with two new items. One for controlling the Toolbar visibility, and another for controlling the Statusbar visibility. And we use the IupConfig to store this selection to be persistent between different application executions.

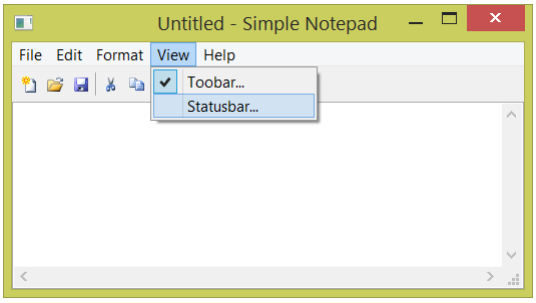
The first change is the inclusion of the function toggle_bar_visibility that handles the changes in visibility in our toolbar and statusbar. When an item in View menu is pressed, if it was checked or, in other words, the bar is visible, set the bar FLOATING attribute to "YES", VISIBLE to "NO" and the item value to "OFF", to hide the bar. If it is not visible, do the opposite. After that, it is necessary to call to **IupRefresh** to recompute the dialog layout.

Next two new callbacks appear: item_toolbar_action_cb and item_statusbar_action_cb. Both callbacks are responsible for calling toggle_bar_visibility and calling IupConfigSetVariableStr to store the item state.

The next change will appear only in main function, and it will be the declaration of our new View submenu and its items, and the new callbacks associations.

Example Source Code [in C] example3_10.c [in Lua] example3_10.lua

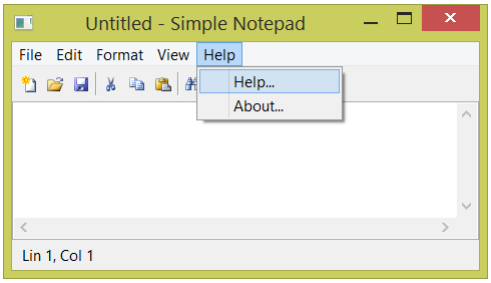




3.11 External Help

One additional feature that our text editor can have is an external help. IUP shows an external help by simply calling the IupHelp function. It will show an Internet browser in the given page, so the application can display some documentation to the user. In our example, it is just a menu item that activates the item_help_action_cb callback that calls the **IupHelp** function. This function shows the IUP website, but it can also show a local HTML file. In Windows, that function is even more flexible allowing opening any kind of document provided that it is associated with an application.

Example Source Code [in C] example3_11.c [in Lua] example3_11.lua



3.12 Final Considerations

That's all for chapter three. If you reached this lines, you are able to build a simple, but fully featured Notepad application using lots of IUP resources.

During this chapter we went from 30 lines of code to 1100 lines (800 in Lua). Only for our simple notepad with file read and write, clipboard access, text search, and other features.

For our final simple notepad code, we have just added two missing features: Replace and Find Next using a hot key. Plus some code organization and comments.

There is still missing features. If we use IupScintilla instead of IupText there is whole new world of possibilities. Any contribution to this code is welcome. Please, send us your comments and suggestions.

Simple Notepad Source Code [in C] simple_notepad.c [in Lua] simple_notepad.lua

In our next chapter we will introduce another Tecgraf library used to draw primitives over a canvas element to build a Paint application.

Previous	Index	Next

4. Simple Paint

4.1 Loading and Saving Images

In the previous chapter, we saw how to build a simple notepad using IUP. In this chapter, we will modify the code presented in the previous chapter and build an application that draws on an image, most like simple paint programs. To do so, we need a structure that represents an image and a few functions that allow us to read and save images in known formats. Therefore we will make use of a library called IM.

IM is a digital images manipulation library. Its main goal is to provide a simple API and abstraction of imaging for scientific applications. In order to use IM in our application, some new includes a are needed: "im.h" which is the main header of IM; "im_image.h" which deals with creation, loading, attribute manipulation and images storage; "im_convert.h" which deals with the conversion among different types of images; and "iup_im.h" which allows the loading and images storing through IUP. Similar to what we did in section 2.1.1 to link IUP's libraries, we will also need to modify the project's link to include the IM (-lim) library and use a C++ linker, even with C code (because internally IM uses C++). More details in section Build Applications of Manual da IM. In Lua, you only need to include two new requires: imlua and iupluaim.

We also added a few functions such as **str_fileext** which extracts the file extension from the filename; **show_error** which creates and exhibits on the screen an error message; and **show_file_error** which uses **show_error** to inform the user what kind of error has happened during the opening of an image file. We also have the **set_file_format**, which sets in what file format the image must be saved based on the extension of the selected file, and the **select_file** which selects a file for reading or saving. The functions related to text manipulation, and some items from Edit menu such as Cut, Del, Find, Replace, Go To, Select All and Format menu were removed, because their use in this application does not make much sense.

The functions **read_file** and **write_file** now use the IM functions, so as the copy and paste items of the Edit menu. The IM functions **imFileImageLoadBitmap** and **imFileImageSave** are entitled to read and save an image from and to a file, using the imImage structure. Note that in the reading, besides the error processing, a test is necessary to convert the image to RGB type, in case it is not of this type yet. This happens because the IM works with several image types and, for the time being and to simplify things, we will adopt the RGB format for the SimplePaint. In writing, we use the same reading format. This format is obtained through an attribute of the image, using the **imImageGetAttribString** function.

In the functions that create a new image, we keep the current image using an IMAGE attribute of canvas, and thus we assign this attribute with a new image and then we destroy the previous image to free-up memory space. The function **set_file_format** was created to treat the format in which the new image must be saved. Usually, we use the same format as the original image. The JPEG format was defined as default, since it is the most popular one.

Another change happened in the clipboard use. The access is done just like in Simple Notepad, but using the NATIVEIMAGE attribute to copy and paste images. This attribute requires a specific format, and for that we used the IUP function called **IupGetImageNativeHandle** , which generates this format from an imImage and vice-versa. Just after the image is read from a file or pasted on the clipboard, we need to redraw the canvas, to do so we call the **IupUpdate** function that will be in charge of calling the redraw callback. Note also that the image on the clipboard could be of any kind, since it was not necessarily copied from this application. Since our application works only with RGB, it may be necessary to remove the alpha channel using **imImageRemoveAlpha**, and convert the format using **imConvertColorSpace**.

Yet, as we save a new file, besides the name of the new file, we need to select an image format that will be defined by the file extension. The **set_file_format** function was created to recover this extension and establish the format in which the new image must be saved. The JPEG format was defined as default, since it is the most popular one.

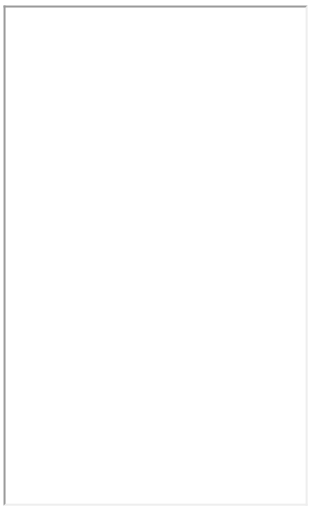
Another interesting novelty was the creation of a **select_file** function that establishes what type of file dialog treatment must be opened. Since dialogs for open file and file saving are very similar, we encapsulated their creation in one single function.

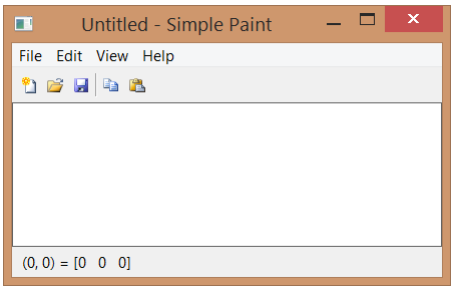
In this example, we also present a new pre-defined dialog called **IupGetParam**, which is used in the File/New to obtain the height and width dimensions of the new image. After this data is obtained from the user, the **imImageCreate** function is called, which creates a new image, with the height and width previously obtained, and of the RGB type, as explained above.

Note that the image drawing is not implemented in this first example, and therefore it will be the object of this tutorial's next item.

In Lua, as mentioned before, the includes are replaced by the imlua and iupluaim requires. As with the IUP, IM is also a Lua package. Its functions are retrieved by the "im." prefix followed by the name of the function without the im present in C. For example: we have im.FileImageLoadBitmap instead of imFileImageLoadBitmap. The functions that in C receive an image as parameter (imRemoveAlpha, imImageDestroy, etc) in Lua are functions of the image itself and are called using ":" (image:RemoveAlpha, image:Destroy, etc), dismissing the image passing as parameter. While Lua has garbage collection, it is also a good practice to call image:Destroy to free-up the memory allocated for the images, since in large applications, which work with several images, the memory consumption could become a problem.

Example Source Code [in C] example4_1.c [in Lua] example4_1.lua





4.2 Drawing with OpenGL

As we saw in the beginning of this tutorial, that IUP is a toolkit for the creation of interface with the user. Although having among its controls a canvas, it does not have functions for drawing on it. For that it will be necessary to include an external library. Among a few options, we choose for this example the OpenGL library for its portability, performance and standardization.

In order to use OpenGL with IUP, besides the GL/gl.h, windows.h (in case you are using windows) and iupgl.h includes, we will need to link with some libraries too. In Windows, opengl32.lib is used, while in Linux the -lGL must be included. The IUP canvas that works with OpenGL is also an additional control called **IupGLCanvas**. A call to **IupGLCanvasOpen** must be included after **IupOpen** so that this control is available.

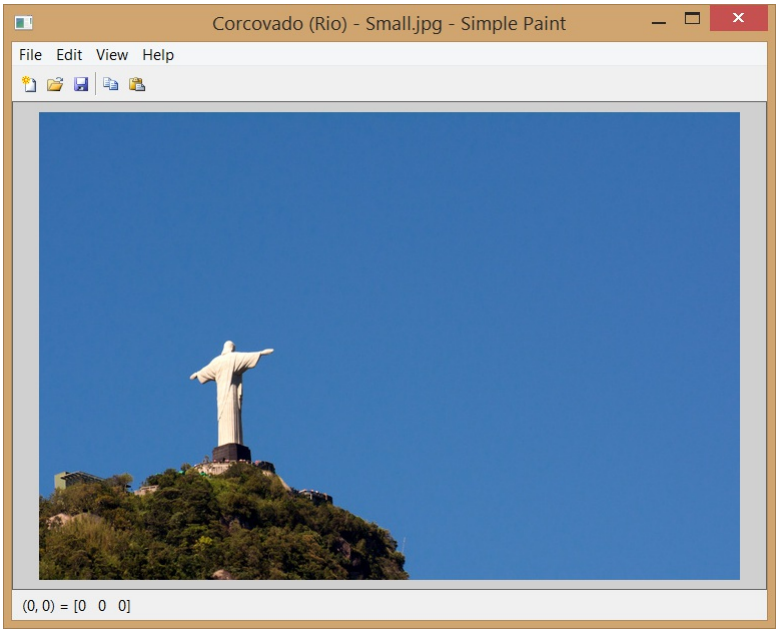
In the code, calls to **imImageGetOpenGLData** in read_file and in new_file, were included to convert the read/created image in an OpenGL compatible format. A canvas action callback was also created. This callback is executed whenever the canvas needs to be redrawn. In this callback, we inform that our GL canvas is the current canvas using **IupGLMakeCurrent**, we start the OpenGL configuration by setting the image alignment to 1, and we adjust the OpenGL coordinates transformation, which by default are between 0 and 1, to between 0 and the canvas size, in a relation of 1 to 1 in pixels. We clean the canvas with the background color using **glClearColor** and **glClear**. Next, we obtain, through the GLDATA attribute, the image data in OpenGL format that needs to be drawn. Then we draw the image in the center of the canvas with **glDrawPixels**. Note that the **glRasterPos2i** and **glDrawPixels** functions do not accept values outside the screen, thus because of this OpenGL limitation, the image to be shown must be smaller than the canvas, or it will not be drawn. We can get around this limitation by using the OpenGL textures support. However, its usage is beyond the scope of this tutorial.

In our example, we are working with double buffer, since we set the BUFFER attribute to DOUBLE in the main function when we created the canvas. This entails that the drawing will be made outside the screen, on a separate buffer. When we finish calling the drawing functions, we show the result by displaying this buffer to the user. To show the result, we call the **IupGLSwapBuffers** function.

In this example, we use another IUP pre-defined dialog called **IupColorDlg**. The **IupColorDlg** is displayed for the selection of a new background color in the View menu. It is a dialog for color selection, and in our example, it changes the canvas background color.

In Lua, to use the OpenGL functions, we use the LuaGL. For this, you only need to require the "luagl" package. To use the **IupGLCanvas** you should also require the "iupluagl". The LuaGL functions follow the Lua standard packages and use the "gl." prefix, for example: "gl.Func" instead of "glFunc" in C.

Example Source Code [in C] example4_2.c [in Lua] example4_2.lua



4.3 Drawing with CD and Printing

In this section, we present an alternative to the OpenGL library. Although having an excellent performance, the OpenGL library has some limitations. There is no support for printing, no metafile output, and it also does not provide support to high quality text. Therefore, many applications need other options. To attend to this other needs, we created the CD library â€" Canvas Draw. You can find this library on: www.tecgraf.puc-rio.br/cd, and it was designed to function together with IUP.

To use it, you need to link with the "cd" and "iupcd" libraries. The "cd.h" and "cdiup.h" includes must appear in the code.

Since we are replacing the OpenGL for the CD, we once again work with the **IupCanvas** control instead of **IupGLCanvas**. We can also remove the libs and OpenGL includes added in the previous section.

In the code, the OpenGL functions calls are no longer needed. **IupGLMakeCurrent** was replaced by **cdCanvasActivate**, **glClearColor** by **cdCanvasBackground**, **glClear** by **cdCanvasClear**, and **IupGLSwapBuffers** by **cdCanvasFlush**.

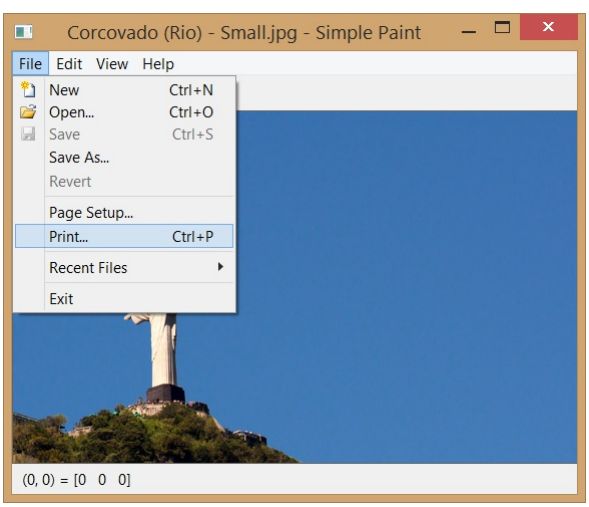
The new MAP_CB e UNMAP_CB callbacks were included. The MAP_CB callback, called **canvas_map_cb**, is responsible for creating the CD canvas using the **cdCreateCanvas**. This is necessary, because to create the CD canvas, the IUP canvas must be mapped beforehand on the native system. One of the parameters that this function receives is the CD_IUPDBUFFER. This informs the CD that it must work on Double Buffer, the same way we have been doing in Open GL. Note that there are now two types of canvas in use - the **IupCanvas** control and the CD library **cdCanvas**. Thus in this callback it is also done an association between these two canvas through a call to **IupSetAttribute**, so it can be retreived later in the action callback. The UNMAP_CB callback named **canvas_unmap_cb** retrieves the CD canvas associated to IupCanvas control and destroys it by calling **cdKillCanvas**. The callback responsible for drawing the image on canvas continues to be the **canvas_action_cb**, the difference is that besides having replaced the OpenGL calls as mentioned before, several of them were removed and replaced for only **imcdCanvasPutImage**. This single call draws an IM image on a CD canvas.

Taking advantage that the CD supports printing, we added a few resources to Simple Paint. Other new callbacks are: **item_pagesetup_action_cb** which is responsible for obtaining from the user, through **IupGetParam**, the height and width of the margin of print preview page; **view_fit_rect** which adjusts the screen to display the entire image; and **item_print_action_cb** which shows the printing dialog.

We also have two new menu items: item_pagesetup e item_print, which call the callbacks with the same name. The print item was associated to a CTRL+P hotkey.

In Lua, the "cdlua" e "iupluacd" requires are necessary. We can remove the "luagl" and "iupluagl" used in the previous example. The CD call function in Lua use the "cd." prefix, as in cd.CreateCanvas instead of cdCreateCanvas and etc. Note that the imcdCanvasPutImage function uses the "im." Prefix, since it belongs to the IM library.

Example Source Code [in C] example4_3.c [in Lua] example4_3.lua



4.4 Interactive Zoom and Scrollbars

In this section, we will add an interactive zoom to our application. To do so, we need to draw the image with a bigger or smaller size than its actual size. This magnification factor needs to be interactively modified by the user through different paths.

Besides that, if the image is bigger than the canvas that we have to draw it, we need a mechanism that allows us to move its visible area. This mechanism is the scrollbar. To enable the scrollbars, we set the SCROLLBAR=Yes attribute of the IupCanvas. But we have to configure them every time the magnification factor is modified and when the application window changes its size. Thus, we need to implement the RESIZE_CB callback of the canvas, so that it calls the **scrollbar_update** function that calculates the scrollbar parameters.

To change the magnification factor, we created a few mechanisms and also added some controls to the statusbar. They are: an **IupVal** which selects a value on a given interval, and three buttons that are responsible for the zoom in, zoom out, and return to original actions. The same buttons' actions can be made through the View menu using item_zoomin, item_zoomout and item_actualsize. We also created hot keys that activate these buttons: CTRL+ (zoom in), CTRL- (zoom out) and CTRL0 to return to original size. Finally, we added a WHEEL_CB callback of the canvas, which is activated through the mouse wheel. In it we used the delta parameter to modify the zoom factor. You can find these news controls in the **create_main_dialog** function, in which we used a shortened way to create a control hierarchy using **IupSetCallbacks** and **IupSetAttributes** together. The result is similar to the creation of controls in Lua. The zoom factor is changed linearly, but its effect is of a power of 2, therefore the controls modify what we call zoom_index between -6 and 6 limits, and the zoom factor is calculated by doing pow(2, zoom_index), which results in a zoom interval of 1% and 6400%. To use the pow function, it is necessary to use the math.h include.

The **scrollbar_update** function performs a very complicated calculation, which is described in the IUP Manual on the **SCROLLBAR** attribute documentation. This is necessary because of the AUTOHIDE attribute, which automatically hides the scrollbar. Notice that in this function, we obtained the canva size in pixels through the RASTERSIZE attribute, and we removed two pixels. This happens because the IupCanvas has the BORDER attribute set as "YES" by default. Thus it is necessary to remove 1 pixel (size of the edge) for the left edge and another for the right edge. The same happens to the superior and inferior edges. We only configure the DX and DY parameters of the scrollbar to equal the visible area of canvas with the magnified zoom. We leave the XMIN, YMIN and XMAX, YMAX parameters with the default values of 0 and 1, respectively. The POSX e POSY attributes inform the shift that the image drawn with zoom must have in order to move according to the scrollbar. Since POSX e POSY are between 0 and 1, this shift in pixels is obtained by multiplying the attribute values for the total size of the image in zoom (view_with e view_height). The **scrollbar_update** function works together with two new functions. When the scrollbar is updated, the **scroll_calc_center** and the **scroll_center** are necessary to keep the image displayed in the same position on the screen while changing the scrollbar configuration.

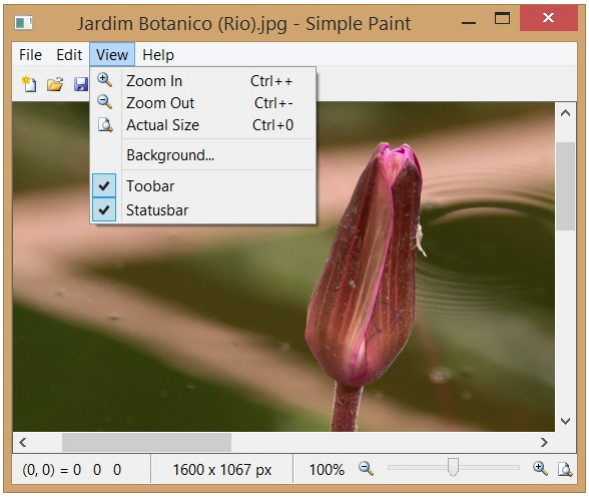
Once we have modified the magnification factor, we need to draw the image. For that, in the action callback of the canvas, we add a new calculation to obtain the position and size of the image to be drawn on canvas. This calculation obtains the zoom factor, resizes the visible area by multiplying by this factor, and repositions the image on canvas.

Since we had been modifying the action, we also added a border around the image using **cdCanvasRect**. We did this because when we include the zoom and scroll, it is usually difficult to locate the image borders, especially if it looks like the background color. Therefore it is common to include an edge around the image to mark the end of it.

We noticed that in example 4_4, we could improve the application state of control when a new image is created. For that, we created the **set_new_image** function that replaces parts of the code in New, Open and Paste. In this function, we encapsulated the change in the dialog title from the file name, and we verified if the image is RGB as described in section 4.3. We also verified if there is a file format for the new image, and if we should adopt the default format. Furthermore, we set the DIRTY value, and finally restarted the zoom factor to normal visualization at 100%.

In Lua, since the mathematical library is already included in the standard parser, there is no need for a new require. Besides the syntax difference among the languages, there is no particular changes.

Example Source Code [in C] example4_4.c [in Lua] example4_4.lua



4.5 Canvas Interaction and a ToolBox

So now we are going to implement a more complex form of interaction with the canvas. We want to have control over the actions of the cursor when moved or clicked over the **IupCanvas**. For that we need two new callbacks: MOTION_CB and BUTTON_CB. Inside these callbacks there will be all the logic behind the interactions we want to implement for a Paint application. But in order to do that we need first to define which type of interaction we want.

In a Paint application the interaction is usually defined by a toolbox where the user chooses a tool to interact with the canvas. The toolbox is a dialog with some special characteristics. In our example code this is done by the create_toolbox function. We are going to reduce the default font size, and we are going to use the TOOLBOX attribute, since we want a dialog with a small foot print on screen. The first thing to notice on its internal controls is the use of an **IupRadio**. All the **IupToggle** inside the radio hierarchy will be mutually exclusive, so when a tool is selected all the others are not selected. We put all the toggles inside a **IupGridBox** so they will be automatically aligned in a rectangular grid with 2 columns. And we are going to need custom images for the tools since they are not available at the **IupImageLib**.

We created those images in a very popular application called Paint.NET. It allowed us to save the RGBA files in the PNG format, then we used the IupView application to convert the files to C source code so we were able to compile them directly inside our application. Another possibility would be to use the **IupLoadImage**, but then our example will have to be able to locate the image files during run time. All the images were processed and its code is pasted at the beginning of the example source code.

After the tools we added a few other controls to support some tools options. Not all tools use all options, so a future enhance to the example would be to hide and show each option accordingly to the selected tool. But for now we are going to leave all the tools options visible all the time. You will see that the tools use controls and features we already described in previous examples. So let's focus on the tools themselves.

The current tool is saved in a custom attribute called TOOLINDEX. We will also use this approach to save the tools options values, such as TOOLWIDTH, TOOLCOLOR, TOOLSTYLE, TOOLFILLTOL and TOOLFONT. This will make the use of these values a lot easier.

We created 10 tools that will use 4 different types of interaction. (1st type) Pointer will use click+drag to also scroll the image that is larger than the visible canvas. (2nd type) Color Picker and Fill Color will just need a click on the canvas. (3rd) Pencil will directly draw over the image using a click+drag approach. (4th) All the shapes (Line, Rect, Box, Ellipse, Oval and Text) will use click+drag to set flags that activates an overlay process in the canvas_action_cb callback, so the tool feedback can be done over the image. When it is done, the final drawing is rendered over the image itself when the button is released. So there were changes to canvas_action_cb, new implementations in canvas_button_cb and in canvas_motion_cb, all working together to implement each interaction.

All these interactions are done while the mouse is pressed over the canvas or when it is simply clicked (pressed+released). There is another type of interaction that uses the concept of a graphical

object over the image. For example, instead of only drawing the feedback while the mouse is pressed, the result creates a graphical object that can be lately modified and manipulated. Just like a selection area in other Paint applications. This implies in a data structure to store the object, and another for the list of created objects. When the mouse is moved near the object, handlers are shown so the user can click and interact with the graphical object. This technique can be use to create a Simple Draw application (like Corel Drawâ„¢), instead of a Simple Paint. Where we will be manipulating vector data instead of raster data, and loading/saving formats like WMF/EMF, SVN, PDF, CGM, DXF, and so on (all supported by the CD library by the way). In terms of user interface features a Simple Draw is most like the same of a Simple Paint application..

The first thing we had to do to implement the interactions was to get that calculation in the canvas_action_cb to obtain the position and size of the image on screen and transform it into a function that we called view_zoom_rect. We are going to need those parameter to convert the coordinates received by the callbacks into coordinates inside the actual image. So in all mouse callbacks, after calling view_zoom_rect we invert the Y axis, because y is top-bottom oriented in IUP, but bottom-top oriented in CD and IM. Then we check if the resulting coordinates are inside the image on screen and convert them to the actual image coordinates using view_zoom_offset. So now (x,y) are inside the image range (0,0)-(image_width-1,image_height-1). In other words we converted screen coordinates into image coordinates.

To actually draw on the image after the interaction we used the CD_IMAGERGB driver pointing to the image data. So we can draw using CD primitives but using the image as the canvas medium. In this way the code becomes very simple an easy to understand. But for text to work properly we must not forget to set the new CD canvas resolution to the same resolution of the screen, so we will obtain a result with the same size in pixels.

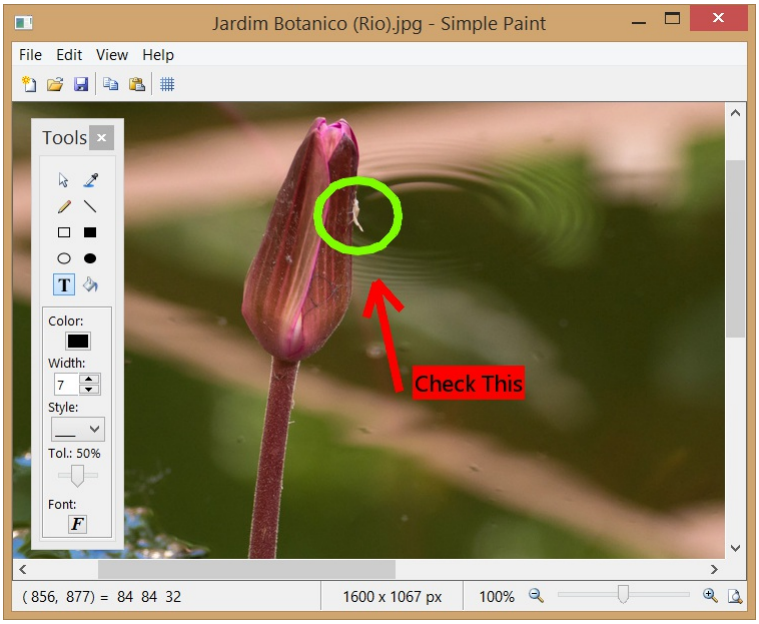
For the Fill Color tool we had to implement a flood fill algorithm. We used a very simple 4 neighbors stack based flood fill. So it is also very didatic. There are several optimizations possible, can you point any?

We also used the canvas_motion_cb callback to update the current pixel color on the Statusbar. This will be done independently of the current selected tool.

For better integration of the main dialog with the toolbox dialog we move the toolbox dialog every time the main dialog is also moved. The MOVE_CB callback of the main dialog is implemented and it will simply offset the toolbox dialog by the same offset the main dialog was moved.

As we are adding layers to the image visualization, we also added a zoom grid feature. It will display a grid over the image when the zoom factor is greater than 200% to help the user to identify pixel boundaries.

Example Source Code [in C] example4_5.c [in Lua] example4_5.lua



4.6 Image Processing and Final Considerations

In our final code for this chapter we are going to add a few image processing functions provided by the IM library. We added a new sub menu to the main menu called "Image", and there we added items for Resize, Mirror, Flip, Rotate, Negative, and Brightness and Contrast. Although all these operations are interesting, we would like you to take a look at the Brightness and Contrast operation. We used a **IupGetParam** dialog as before, but this time we implemented the PARAM_CB dialog callback that allow us to interactively update the image while changing the operation parameters in the dialog. So helping the user to find the best combination for those parameters for the desired result. For this to work we are going to temporarily replace the current image with the processed image, and simply update the canvas. The result is very effective. IM has lots of other image processing operations that we will let you to explore, to use them we need to link with the **im_process** library too. This will also allow us to replace the flood fill and fill with white routines by **imProcess** versions.

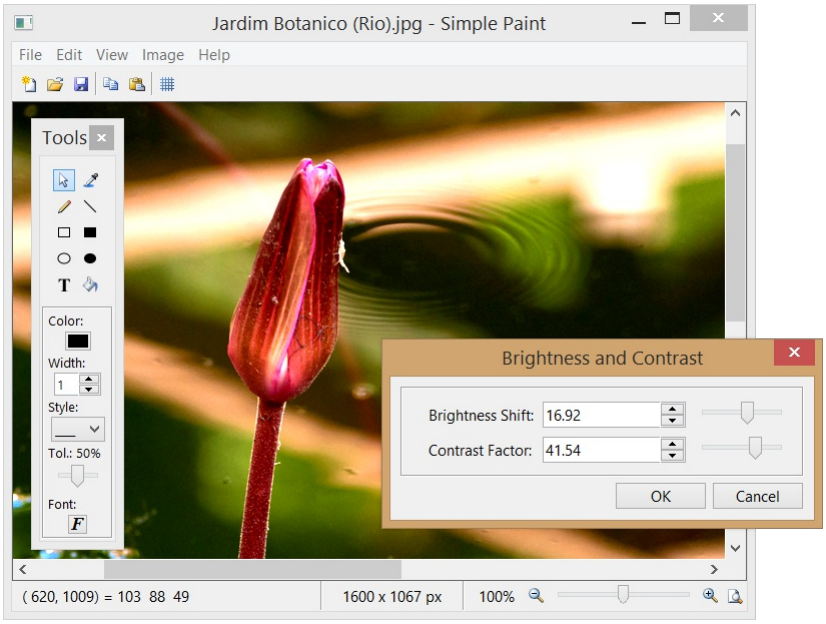
Since we are adding utilities libraries, let's also include the **cdim** library that will allow us to replace the CD_IMAGERGB driver by the CD_IMIMAGE driver, and the **imcdCanvasPutImage** macro by the **cdCanvasPutImImage** function. Providing a more elegant code for our final version.

And we are done for this chapter. We went from 800 lines, almost all based on our previous example, to 2500 lines. Implementing a fully featured Paint application using IUP and CD resources. Which include loading and saving of image files, drawing and printing of images, zoom and scroll support, and the most important, how to interact with a canvas and its drawing in several ways.

Still there are always possible enhancements such as using the IM Video Capture features to obtain an image from a camera, Undo/Redo support using a stack of images, transparency using an alpha component in color, area selection...

The toolbox is also an interface element that can have several approaches. Instead of jut hiding it we could use **IupDetachBox** to insert it on the main dialog at the left side of the canvas, so it can has 3 states: hidden, floating as a dialog, and attached just like the toolbar. And when attached there is also another possibility, we could use an **IupExpander** so we can dynamically show and hide its contents leaving a direct affordance in the dialog to do that.

Example Source Code [in C] simple_paint.c [in Lua] simple_paint.lua



In our next chapter we will introduce some advanced techniques for IUP applications.

Note: both images used for the screen shots are Copyright © Antonio Scuri, and distributed under the Creative Common License.

Previous	Index	Next

5. Advanced Topics

5.1 C++ Encapsulation

As you recall from chapter 4, our Simple Paint source code now has 2500 lines. It is a lot to process, specially if you are looking for bugs, or learning how it works without reading all the previous examples that evolved into the final code. So it is time to use some software engineering techniques to improve quality and maintenance. We can do that in C too, but modern applications are more and more using C++, also because the language provides some tools to easy that task. So the first thing we are going to do is to convert the code from C to C++.

Actually if you simply save the "simple_paint.c" as "simple_paint.cpp", and use a C++ compiler it will fully work. But it is not what we meant. We would like to isolate parts of the code to reduce the interference of one part in another. This is called encapsulation in software engeneering. The simplest way to do that in C++ is to use classes for major features in the application. So looking at our code we can see at least 3 groups of functions: the main dialog, the toolbox dialog, and file management.

So we decided to start with 3 classes: **SimplePaint** (the main dialog), **SimplePaintToolbox** (the toolkbox dialog) and **SimplePaintFile** (file management). If you compare the C and the C++ codes they are very similar, except that functions are now class methods. Even IUP callbacks are now methods, but there is a catch here. Class methods can NOT be used as function pointers as the ones used by **IupSetCallback**. It order to be able to do that we must implement a static method, use it as the callback, and from inside that code call a class method. To do that every time for all the callbacks can be very task consuming. So we created a few macros to help implementing callbacks as class methods.

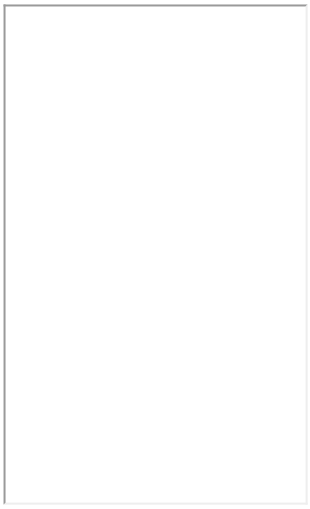
These macros are available in the "iup_class_cbs.hpp" include file. To use the macros for the callbacks you must call the IUP_CLASS_INITCALLBACK(ih, class) macro once, usually in the class constructor after the IUP dialog was created. This macro will register the IUP element so the class object can be retrieved later transparently for the application. The macros however can be called in any order. So we will use the IUP_CLASS_DECLARECALLBACK_*(class, callback) macros to declare the callbacks as methods. Since we have several different callbacks because of the different parameters, there are several different macros, one for each callback signature found in IUP elements. The static method has the same name of the callback used in the macro with a "CB_" prefix. So you can also directly use its name in **IupSetCallback** if necessary. To actually set the callback of an element simply call IUP_CLASS_SETCALLBACK(ih, name, callback) just like you call **IupSetCallback**. The callback name will be the same you used in IUP_CLASS_DECLARECALLBACK_*(class, callback), in fact it will simply call **IupSetCallback** with the static callback using the "CB_" prefix. So here is a simple class to illustrate this procedure:

```
class SampleClass
{
  int sample_count;

public:
  SampleClass()
  {
    sample_count = 0;

    Ihandle* button = IupButton("Inc", NULL);
    // 2) Associate the callback with the button
    IUP_CLASS_SETCALLBACK(button, "ACTION", ButtonAction);

    Ihandle* dialog = IupDialog(button);
    // 1) Register this object as a callback receiver (only once)
    IUP_CLASS_INITCALLBACK(dialog, SampleClass);

    IupShow(dialog);
  };

protected:
  // 3) Declare the callback as a member function
  IUP_CLASS_DECLARECALLBACK_IFn(SampleClass, ButtonAction);
};

// 4) Define the callback as a member function
int SampleClass::ButtonAction(Ihandle*)
{
  sample_count++;
  return IUP_DEFAULT;
}
```

Using these macros methods of the same C++ class can be set as callbacks for any element. For clarity we are going to use a single class to process the callbacks of all elements that are children of the same dialog. So we will need 2 classes for our 2 dialogs. The third class will handle only the image file management and it will not have callbacks. To be able to isolate the toolbox from the main dialog class we will need some extra methods that will operate over the toolbox dialog. Apart from that all the methods of both dialog classes are directly equivalent of a function in our C source code. We also tried to maintain their position in the source code to simplify the comparison between the two.

Notice that only a few methods are left public in both classes, that is where the encapsulation occurs.

Example Source Code [in C++] simple_paint1.cpp [in C] simple_paint.c



5.2 C++ Modularization

In the previous section we purposely left all classes in the same file so you will be able to compare it with the C source code. But now is the time to split the code in several modules, one for each class. So instead of a 2500 lines file, we now reduce to 1000 lines for the main file (where the main dialog is) and the rest distributed in the other files.

But we actually were able to create a total of 6 modules! 3 modules for the classes we already described, and 3 new modules. The 3 modules we already expect are: "simple_paint.cpp/h" (**SimplePaint** class), "simple_paint_toolbox.cpp/h" (**SimplePaintToolbox** class) and "simple_paint_file.cpp/h" (**SimplePaintFile** class).

The first new module is a very simple one, called "simple_paint_main.cpp". It contains only the "**main**" function necessary for the application starting point.

The second new module is to store the utility functions that are not related to any specific classes. It is called "simple_paint_util.cpp/h".

And the third module is a new class that we identified mixed up with our **SimplePaint** class. Inside the main dialog we have a very important control that does the most important interface task that is to show the image and do the direct interaction defined by the toolbox. As you can guess now is the **IupCanvas**. It has several speciall callbacks and as we said is the essential tool for our paint interface. So it is a natural candidate for separate class we called **SimplePaintCanvas**. It will hide the canvas from the main dialog, and encapsulate all its features, mainly interaction and zoom control. It is called "simple_paint_canvas.cpp/h".

Now it is not just easier to find the part of the code you want to change, but it helps to reduce the interference in other modules of what you have just changed.

But notice that our modularization is still not perfect. Inside **SimplePaintCanvas** there are some references to controls that are located in the Statusbar.

The next stage will be to use Dynamic Dispatch, or in C++, virtual methods and inheritance to implement classes for the interactive tools to make it easier to add new tools.

Also our classes are instantiated just one time. What about adding support for editing multiple image files simultaneously, but instead of using the old Windows MDI concept to use a **IupTabs** to alternate between the files?

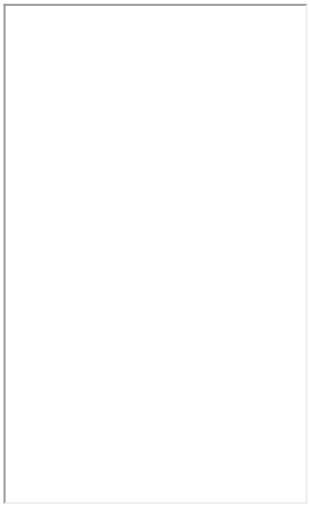
In Lua the changes would be very similar, using tables to isolate the code of each module.

So there are plenty of possibilities for improving our object oriented modeling. If you implement some of them, please let us know and share your code so we can add it to the tutorial.

Example Source Code (Implementation) [C++] simple_paint.cpp simple_paint_canvas.cpp

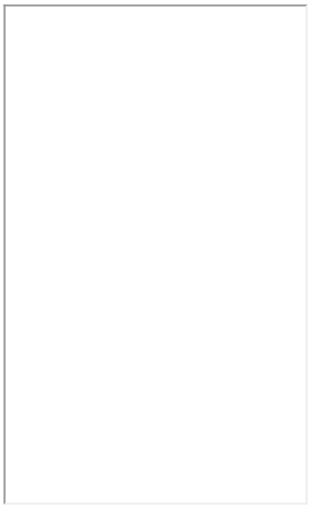
simple_paint_toolbox.cpp simple_paint_file.cpp simple_paint_util.cpp

simple_paint_main.cpp



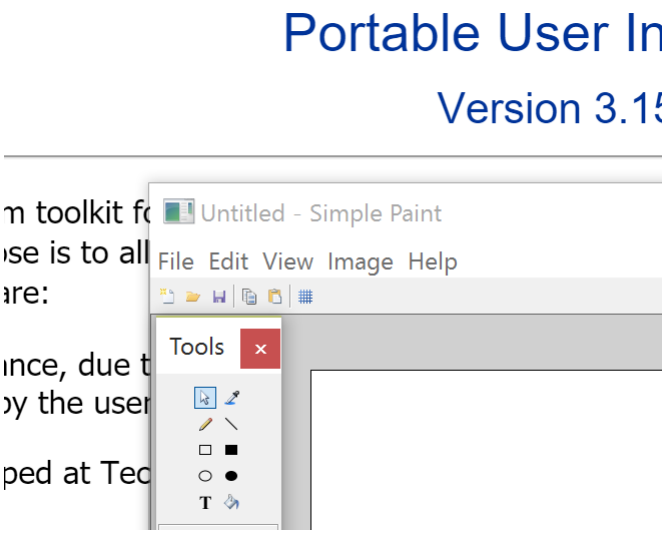
Example Source Code (Declaration) [C++] simple_paint.h simple_paint_canvas.h

simple_paint_toolbox.h simple_paint_file.h simple_paint_util.h

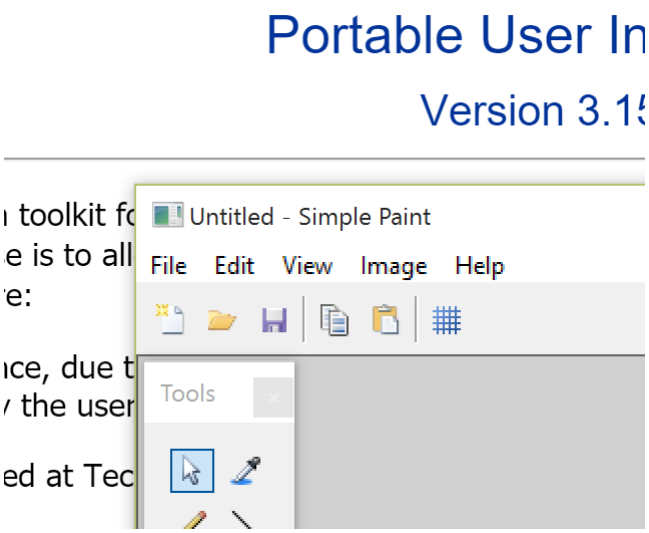


5.3 High Resolution Display

During the 90's the 15" monitors with 1024x768 pixels were the most popular graphic resolution was about 85 DPI. But that was a long time ago. Soon we started to see 19" monitors with 1280x1024 pixels and 92 DPI. Later the most popular are the 20" Full-HD wide screen monitors (16x9 at the same height as 19" standard 4:3 monitor) with 1920x1080 pixels and a resolution of about 96 DPI. Finally we got to the 4K wide screen monitors with 3840x2160 pixels. Even for a 24" wide screen monitor, the resolution 3840x2160 pixels is 186 DPI. It is a lot more than 96 DPI. For instance 16x16 pixels icons are very tiny. Here is an example using a browser page as reference:



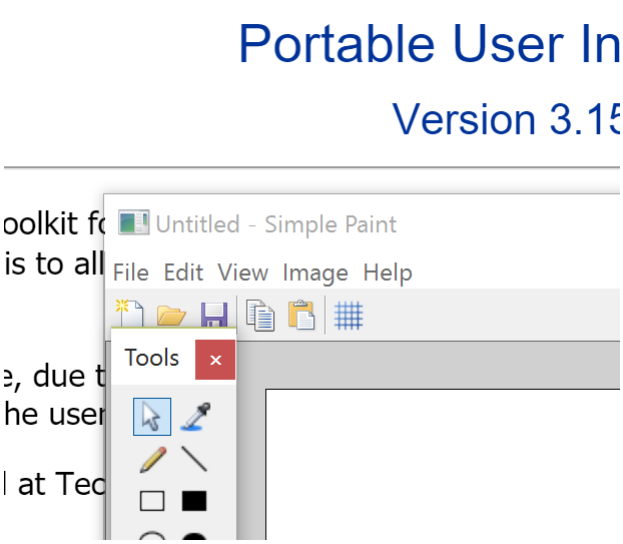
To support such high resolution the application should be able to compute its layout using a larger font, and to include image sets for button when in high resolution. Usually this not occurs. IUP will automatically take care of the layout (considering that the application used SIZE, CMARGIN and CGAP, instead of RASTERSIZE, MARGIN and GAP), but images are commonly provided in one size only, our **SimplePaint** is no different. To compensate that Microsoft used a strategy in Windows that will do a low level resize of the application, so it can improve its readability in sacrifice of its resolution. So this is how our SimplePaint looks like with the resize strategy:



The **IupCanvas** will report a size that is actually smaller than the size on screen, because the application will be run as if in a Full-HD monitor, not using the available resolution. Notice that even the menu text is blurred. To avoid that we added a few lines to the Manifest file, declaring that we are a **dpiAware** application. So this was how we obtained the previous screen shoot with the small icons. Now that we know how to avoid the Microsoft resize strategy it is time to improve our application readability by our own.

We used in SimplePaint two sets of images, one from the **IupImageLib** stock images, and one created just for **SimplePaint** which are only 16x16 pixels. We actually don't have to worry about the stock images, because since IUP 3.16 they are automatically resized accordingly the screen resolution (to obtain the first screen shot, we also had to disable this feature). But the SimplePaint toolbox images only have 16x16 pixels. The solution would be to add new images with at 32x32 pixels that can be used in place of the 16x16 if the resolution is very high, for instance by checking

the global attribute SCREENDPI being greater than 150 DPI and selecting 16x16 or 32x32 images. But IUP also provides an automatic resize for images using the same strategy that Microsoft does for the whole application. To do that we set the IMAGEAUTOSCALE global attribute to the "DPI" value. So all the images will be scaled accordingly. The result is the following:



Now we have the best of both worlds, high resolution with normal readability.

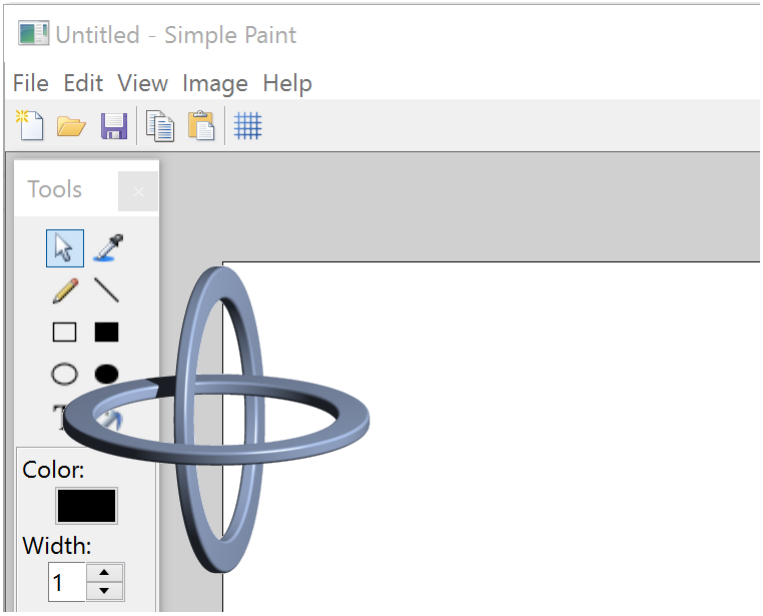
5.4 Splash Screen, About and System Information

Sometimes the application take a long time to start up, maybe it has many things to initialize for instance. One common approach to distract the user while this initialization occurs is to use a Splash screen. It is a dialog shown while the application initializes that usually has a decorative purpose only. So many companies use it to show the application logo and/or the company logo as a marketing strategy. Although we can use a normal dialog to do that, this dialog will have no decorations and no controls inside. We are going to use a single image as contents, and a trick to show it with a transparent background. The trick is the OPACITYIMAGE attribute of the **IupDialog**, it will use the transparency of the image to create a mask for the dialog shape, so the dialog will be shown with a non rectangular area, on top of what's on the background.

But we would like that to take two moments, first when the logo is shown alone, and a second moment when the logo is shown with our application on the background, while it initializes. To control the timing we use an **IupTimer** set initially for 1 second, then hold the execution, and inside the timer callback restart the timer for another second but now letting the application initializes normally. The splash dialog will be automatically destroyed at the end of the second moment.

The image we chose is the Tecgraf logo with 317x317 pixel. It is a large image to convert it to a C source code and embed into the application executable just for the splash screen. So we are going to load it from its file during run time. We could directly use IM functions to do it, and use CD to draw it, but we don't need much control over the drawing this time, so a simpler way is to use the **IupLoadImage** utility function that loads a file and returns an **IupImage** ready for IUP controls. The problem is that we now have to distribute our application with the logo file, and in run time locate that file for loading. There are many strategies to do that, we decided that our logo will be located in the same folder of the executable or in the parent folder, so we use the global attribute "EXEFILENAME" that contains the executable file name with full path (notice that the **main** function argument "**argv**[0]" not always contains the full path). From it we extract the path where it is located so we can concatenate with the logo file name. If the file is not found the splash is simply not shown.

Example Source Code [in C++] simple_paint_splash.cpp



So now that we have a handsome logo, why not to improve our About dialog too? The About dialog also has an important job to show the application version. Until now we didn't have a version number, but our **SimplePaint** is getting more complex every day, so we must be able to know which version we are running. If we count from the start, in chapter 4 we should have reached version 1.0. In chapter 5 we moved to C++ but didn't actually added new features, so it was version 1.1. So for our splash screen commemorative edition we will simply define it as version 1.2.

We also added some company information and a contact e-mail. In the screenshot bellow the sharp eye will notice a caret in front of the e-mail text. That's because it is not an **IupLabel**. It is an **IupText** without borders and with the same background of the dialog. We use it instead of a label so the user can select the text and copy to the clipboard to paste it somewhere else. Another option would be to use the **IupLink** element with the text "mailto:iup@tecgraf.puc-rio.br", this will invoke the system e-mail application when clicked.

Finally there is also a button for System Information that shows a pre-defined dialog called **IupGetText** with textual information about the current system that can also be copied to the clipboard for use in error report for instance. Notice that all the system information were obtained from IUP global attributes.

Here is a sample of the System Information text:

```
-------  System Information  -------

IUP 3.15 Copyright (C) 1994-2015 Tecgraf/PUC-Rio.

  System: Win10
  System Version: 10.0.10240 (x64)

  Screen Size: 3840x2080
  Screen Depth: 32

IM 3.9.1 Copyright (C) 1994-2015 Tecgraf/PUC-Rio.

CD 5.8.2 Copyright (C) 1994-2015 Tecgraf/PUC-Rio.
```

5.5 Dynamic Libraries

Now it is time to distribute our application. There are many installation creation tools for Windows like Microsoft Windows Installer (Free - defines the MSI package format), Install Shield (Commercial), Nullsoft NSIS Installer (Free - own package format), Inno Setup (Free - own package format), and WiX toolset (Free - can produce MSI packages). Here is a simple comparison on **StackOverflow**: Free Install Wizard Software. In Linux it is very common to distribute only the source code, but this becomes more complex when it involves libraries that are not installed on the system, like IUP, CD and IM. To build distribution packages for Linux search for rpm and deb package formats on the Web.

But before creating and installer we need to define which files we will distribute.

When using **static libraries** to link our application, we simplify the deployment because everything is packed in a single file. But in link time we have to know all dependencies of all libraries we are using, and still this does not guaranties that some library will have an external dependecy. Use know we use 3 libraries IUP, CD and IM. The main IM library (**im**) depends on the Zlib library (**zlib1**), and must not forget the IM Image Processing library (**im_processes**) that contains the functions we use in section 4.6. The CD main library (**cd**) depends on the FreeType library (**freetype6**) which in turn depends also on the Zlib library, finally the CD main library in Windows depends on the GDI (**gdi32**) and in Linux depends on the GDK and Cairo libraries. The main IUP library depends on GTK in Linux and USER (**user32**), Common Dialogs (**comdlg32**) and Common Controls (**comctl32**). To use IUP and CD together we need also the CD_IUP library (**iupcd**). To use IUP and IM together we need the IUP-IM Utilities library (**iupim**). And the IUP Image Library (iupimglib) for the stock images. So our actual link list will include:

```
iupimglib iupcd iup cd freetype6 im_process im zlib
plus in Windows: comctl32 comdl32 user32
plus in Linux: gtk-x11-2.0 gdk-x11-2.0 gdk_pixbuf-2.0 pango-1.0 pangox-1.0 gobject-2.0 gmodule-2.0 glib-2.0
```

The executable takes longer to link and all exported functions of all static libraries must not have a single function with the same name. And they all must use the same C Run Time Library when compiled, mixing different run times can have unpredictable results and usually linker errors.

On the other hand, when using **dynamic libraries** things get more simpler when developing and carefully when distributing. This time we do not have to know all the dependencies, only the direct dependencies used by our own code. So when linking we will specify only the following libraries:

```
iupimglib iupcd iup cd im_process im
```

The executable will link much faster and there will be much less room for conflicts. If you keep the memory allocation and release isolated by each library, meaning what allocated in IUP is released by IUP, what's allocated in CD is released by CD, and so on, then there will be no C Run Time library memory problems, even when using libraries that were linked with different Run Time libraries (standard structures like FILE* are also non interchangeable). But now we have to include all those libraries and their dependencies in the distribution package. The fist time you build the distribution package is problematic because you have to make sure that you get everything you will need in a foreign system. In Windows a very useful application in the Dependency Walker (Free). It will list all the DLLs linked to the application and their respective dependencies (don't forget to include the C Run Time DLL too, usually "msvcrXX.dll"). In Linux you can use the "ldd" application and in MacOSX the "otool" application for that purpose, but they are more limited.

In Windows, when running an application that depends on DLL is quite simple, if you copy the DLLs to the same folder of the application the system will automatically locate the DLLs. There is no need to change the PATH or to copy the DLLs to the Windows/System folder. In Linux, if you do not copy the .so files to the system folder then you need to at least set an environment variable called LD_LIBRARY_PATH (DYLD_LIBRARY_PATH in MacOSX) to include the folder were the dynamic libraries are. For example:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/tecgraf/simple_paint
```

So we actually moved the problems from one place to another, although it is still more interesting to work with dynamic libraries since they isolate a library code from another, linker is faster, and individual updates to the dynamic libraries are far more simple to deploy.

Previous	Index	Next

System

IUP has several global tables as together with some system tools must be initialized before any dialog is created. And the IupLua binding must be initialized also.

The default system language used by predefined dialogs and messages is Portuguese. But it can be changed to English.

System Guide

Initialization

Before running any of IUP's functions, function **IupOpen** must be run to initialize the toolkit.

After running the last IUP function, function **IupClose** must be run so that the toolkit can free internal memory and close the interface system.

Executing these functions in this order is crucial for the correct functioning of the toolkit.

Between calls to the **IupOpen** and **IupClose** functions, the application can create dialogs and display them.

Therefore, usually an application employing IUP will have a code in the main function similar to the following:

```
int main(int argc, char* argv[])
{
  if (IupOpen(&argc, &argv) == IUP_ERROR)
  {
    fprintf(stderr, "Error Opening IUP.")
    return;
  }

  ...
  IupMainLoop();
  IupClose();

  return 0;
}
```

IupOpen

Initializes the IUP toolkit. Must be called before any other IUP function.

Parameters/Return

```
int IupOpen(int *argc, char ***argv); [in C]
[There is no equivalent in Lua]
```

argc and **argv**: are the same as the application "main" function function. Some parameters processed by the driver can be removed so the address is necessary. They can be NULL. (Since 2.7)

Returns: IUP_OPENED (already opened), IUP_ERROR or IUP_NOERROR. Only in UNIX can fail to open, because X-Windows may be not initialized.

Notes

In Windows, **CoInitializeEx(COINIT_APARTMENTTHREADED)** and **InitCommonControlsEx(ICC_WIN95_CLASSES)** functions are called.

In Motif, **XtOpenApplication** function is called.

For a more detailed explanation on the system control, please refer to Guide / System Control.

Environment Variables

The toolkit's initialization depends also on platform-dependent environment variables, see each driver documentation.

QUIET

When this variable is set to NO, IUP will generate a message in console indicating the driver's version when initializing. Default: YES.

VERSION

When this variable is set to YES, IUP generates a message dialog indicating the driver's version when initializing. Default: NO.

Lua Binding

This function should be called by the host program and before the IupLua initialization function **iuplua_open**. If not the IupLua initialization function will call it.

See Also

iuplua_open, IupClose, Guide / System Control

IupClose

Ends the IUP toolkit and releases internal memory. It will also automatically destroy all dialogs and all elements that have names.

Parameters/Return

```
void IupClose(void); [in C]
iup.Close() [in Lua]
```

Notes

In Windows, the CoUninitialize function is called.

In Motif, the XtDestroyApplicationContext function is called.

This function is usually called by the application. But if IUP is dynamically loaded in Lua 5 then you should call **iup.Close** from Lua.

See Also

IupOpen

iuplua_open

Initializes the Lua Binding. This function should be called by the host program before running any Lua functions, but it is important to call it after **IupOpen**.

It is also allowed to call **iuplua_open** without calling **IupOpen**. Then **IupOpen** will be internally called. This is also valid for all the additional controls when IUP is dynamically loaded. To call **IupClose** in this way you must call **iuplua_close**.

This enable you to dynamically load IUP using Lua 5 "require".

Parameters/Return

```
int iuplua_open(lua_State *L); [in C]
[There is no equivalent in Lua]
```

Returns: 0 (the number of elements in the stack).

Notes

For a more detailed explanation on the system control for the Lua Binding, please refer to System Guide.

See Also

IupOpen, Guide / System Control

IupVersion

Returns a string with the IUP version number.

Parameters/Return

```
char* IupVersion(void); [in C]
iup.Version() -> (version: string) [in Lua]
```

```
int IupVersionNumber(void); [in C]
iup.VersionNumber() -> (version: number) [in Lua]
```

Returns: the version number including the bug fix. The defines only includes the major and minor numbers. For example: "2.7.1".

Definitions

```
[in C]
IUP_NAME            "IUP - Portable User Interface"
IUP_COPYRIGHT       "Copyright (C) 1994-2011 Tecgraf/PUC-Rio."
IUP_DESCRIPTION     "Multi-platform toolkit for building graphical user interfaces."
IUP_VERSION         "3.5"
IUP_VERSION_NUMBER  305000
IUP_VERSION_DATE    "2011/04/26"

[in Lua]
iup._NAME
iup._DESCRIPTION
iup._COPYRIGHT
iup._VERSION
iup._VERSION_NUMBER
iup._VERSION_DATE
```

Motif System Driver

Driver for the X-Windows/Motif 2.x environment.

Environment Variables

DEBUG

This variables existence makes the driver operate in synchronous mode with the X server. This slows down all operations, but allows immediately detecting errors caused by X.

Default Values Resource Files

Some default values used by the driver, such as background color, foreground color and font, can be set by the user by means of a resource file called "Iup". It must be in the users home or in a directory pointed to by the XAPPLRESDIR environment variable. Below you can see an example of this files contents:

```
*background: #ff0000
*foreground: #a0ff00
*fontList: -misc-fixed-bold-r-normal-*-13-*
```

The values used in the example above are the ones used by IUP if these resources are not defined.

Also a resource file named ".Xdefaults" will also affect the visual appearance of all applications that use Motif and Intrinsics.

Tips

Dynamic Libraries in Linux

When using dynamic libraries in Linux, the "libiup.so" uses the GTK driver for newer systems (Linux26g4). So applications that dynamically load IUP will always use the "libiup.so", for example Lua using require. To use the IUP Motif dynamic library in Linux you must rename the "libiupmot.so" to "libiup.so", so the Motif driver will be loaded instead of the GTK driver.

In older systems (<=Linux26 with gcc 3) the "libiup.so" already contains the Motif driver.

During linking in the Solaris environment: Can not find libresolv.so.2

This error occurs if the system does not have an applied patch containing this library.

This library is important for all installations of Solaris 2.5 and 2.5.1 (SunOS 5.5 and 5.5.1, respectively). It is a correction of the DNS system, involving security.

The web address to get these patches is SunSolves http://sunsolve1.sun.com/sunsolve/pubpatches/patches.html. Select the Solaris version you wish (2.5 or 2.5.1 for Sparc) and download the patches 103667-09, 102980-17, 103279-03, 103708-02, or more recent for 2.5 (the number after the - is the patch version, and the more recent number is the patch), or 103663-12, 103594-14, 103680-02 and 103686-02 for 2.5.1. All of them have a README file explaining installation, and groups have to be installed together.

TrueColor canvas

Whenever a canvas is created, one tries to create it with a TrueColor resolution Visual. This is not always possible, since it is subject to many conditions, such as hardware (graphics board) and the X servers configuration.

The **xdpyinfo** program informs which Visuals are available in the X server where the display is being made, so that you can see if your X allows creating a canvas with a TrueColor Visual. In some platforms, however, the X server may not make a TrueColor Visual available, even though the graphics board is able to display it. In this case, restart the server with parameters that force this. Below is a table with such parameters to some systems where the IUP library has been tested. If the command does not work, or if it is not possible, then the graphics library really does not support 24 bpp.

System	Execution Command
Linux	startx --bpp 24
AIX	(not necessary)
IRIX	(not necessary)
Solaris	(not necessary)

Since color requests are always successful in TrueColor/24bpp windows, we have minimized visualization problems for images that make use of complex color palettes (when there is a high color demand, not always all colors requested can be obtained). The IUP applications also coexist more peacefully with other applications and among themselves, since the colors used by TrueColor/24bpp windows do not use the colormap cells used by all applications.

XtAddCallback failed

When a warning about XtAddCallback appears during the application initialization, and it aborts, this usually means that you are using a Motif with a different version than the Motif used to build IUP. Reinstall Motif or rebuild IUP using your Motif.

Indigo Magic look in Sgi

To turn on the Indigo Magic look for an application, simply set the applications sgiMode resource to TRUE. Typically, you should add this line to the "/usr/lib/X11/app-defaults" file for your application:

appName*sgiMode: TRUE

where appName is the name of your application.

Win32 System Driver

This driver was designed for the modern Microsoft Windows in 32 bits or 64 bits (XP/2003/Vista/7).

DLL

To use DLL, it is necessary to link the application with the IUP.lib and IUPSTUB.lib libraries (for technical reasons, these libraries cannot be unified). Note that IUP.lib is a library specially generated to work with iup.dll, and is usually distributed in the same directory as iup.dll. the IUP DLL depends on the MSVCRT.DLL, that it is already installed in Windows.

For the program to work, IUP.dll must be inside a PATH directory. Usually the program does not need to be re-linked when the DLL is updated.

Tips

Dialog Contents Zoomed by the System

In Windows 8.1, Microsoft introduces a feature to support High DPI screens. If your application does not declares it is DPI aware, and the user configure the screen resolution for values greater than 100%, Windows will report a resolution of 96 DPI for the application and it will scale the contents of the dialog accordingly to the scale factor (actual resolution in DPI/ 96). To avoid that include the Manifest file when building the executable. The "iup.manifest" file in the "iup/etc" folder already contains the necessary changes to declare the application DPI aware.

Text will look better if a high DPI setting is used. For standard monitors (1280x1024) you can use 120 DPI (125%), for FullHD (1920x1080) can use 144 DPI (150%), and for 4k (3840x2160) can use 192 DPI (200%). To be able to respond to these resolution changes the application should avoid using sizes in pixels (like: RASTERSIZE, GAP, MARGIN) use character sizes instead (like: SIZE, CGAP, CMARGIN).

But there is a problem with the images. Usually applications use 16x16 images for toolbar buttons, in a 4k resolution this looks very small, so ideally the application should have at least 3 sets of buttons: 16x16, 24x24 and 32x32. We recommend using 24x24 images for toolbar buttons when the SCREENDPI (global attribute) is greater than 120 DPI (> 125%), and 32x32 when greater than 144 DPI (> 150%). If that is not possible the global attribute IMAGEAUTOSCALE and the **IupImage** attribute AUTOSCALE can be used to automatically scale images with BPP > 8. Stock image size is already automatically selected and resized if necessary.

UTF-8

When IUP is built with UNICODE enabled, it is possible to specify strings in UTF-8. But the default is still to use the current locale. To use UTF-8 strings set the global attribute UTF8MODE to YES.

LINK : warning defaultlib 'LIBCMT' conflicts with use of other libs; use /NODEFAULTLIB:library

This is a message displayed by Visual C++ compilers when one or more libraries included for linking is not using the same C Run Time Library as the application. In the same Visual C++ compiler there are 4 different libraries resulting from the combination of 2 options: **debug**/**release** x **dll**/**static**.

The default configuration when a new project is created uses the C Run Time Library in a DLL, options named "Multi-threaded Debug DLL (/MDd)" for the **Debug** configuration and "Multi-threaded DLL (/MD)" for the **Release** configuration. The IUP package that matches that configuration is the "dll*" packages.

If you want to use static libraries then use the options "Multi-threaded Debug (/MTd)" for the **Debug** configuration and "Multi-threaded (/MT)" for the **Release** configuration. Then use the IUP packages named "vc*".

The IUP pre-compiled packages do not have debug information, so even selecting the correct **dll**/**static** combination, the warning will also be displayed. In this case the warning is harmless. But if you really want to avoid the warning simply use the same option without the Debug information for Release and Debug configurations.

Finally one thing that is **NOT** recommended is to do what the linker warning suggests, to ignore the default libraries using the /NODEFAULTLIB parameter. Only use that parameter if you really know what you are doing, because using it you can create other linking problems.

COM Initialization

IupOpen calls "CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);", if you need another concurrency model call CoInitializeEx with other options before calling **IupOpen**. Be aware that some features in some controls require single-thread apartment, and they will stop working, this includes: **IupFileDlg** when selecting a folder, and **IupOleControl**.

InitCommonCtrlEx Linker Error

On Windows a common error occurs: "Cannot find function **InitCommonCtrlEx**()" This error occurs if you forgot to add the comctl32.lib library to be linked with the program. This library is **not** usually in the libraries list for the Visual C++, you must add it.

Custom IupFileDlg

To use some cursors and the preview area of **IupFileDlg** you must include the "iup.rc" file into your makefile. Or include the contents of it into your resource file, you will need also to copy the cursor files.

Windows XP/Vista/7/... Visual Styles

Windows Visual Styles can be enabled using a manifest file. Uncomment the manifest file section in "iup.rc" file or copy it to your own resource file (you will need also to copy the manifest file "iup.manifest" or "iup64.manifest").

When using Visual C++ 8/9/10/... with a manifest file, configure the linker properties of your project to do NOT generate a manifest file or the Windows Visual Styles from the RC file won't work. Also when using Visual C++ 8/9/10/... you can avoid using the manifest by using the following pragma on your code:

```
#pragma comment(linker,"\"/manifestdependency:type='win32' \
name='Microsoft.Windows.Common-Controls' version='6.0.0.0' \
processorArchitecture='*' publicKeyToken='6595b64144ccf1df' language='*'\"")
```

If your Windows is using the Windows Classic theme or any other theme, IUP controls appearance will follow the system appearance only if you are using the manifest. If not using the manifest, then it will always look like Windows Classic.

Help in CHM format fail to open

When you download a CHM file from the Internet Windows blocks your access to the file. You must unblock it manually. Right click the file in Explorer and select "Unblock" at the bottom of the dialog.

Visual C++ 6

Since 3.0 Visual C++6 is not supported, although we may still provide pre-compiled binaries. To compile the IUP 3 code with VC6 you will need to download a new Platform SDK, because the one included in the compiler is too old. But it cannot be a too new one also, because the compiler will report errors in the newest headers.

We recommend you to upgrade your compiler. Visual C++ Express Edition is a free compiler that has everything VC6 had and more.

GTK System Driver (since 3.0)

This driver was designed for the GTK+ version 2 and 3. It can be compiled in Windows or UNIX.

Although GTK has layout elements they are not used. IUP fill, vbox, hbox and zbox containers are implemented independent from the native system.

The oldest GTK version that can be used is 2.4, oldest versions will not compile. But using versions older than 2.12 several features will not work. Critical features need at least version 2.8.

Currently it is not available for IRIX, AIX and SunOS. But is it available for SunOS10 and it is not available for Linux24.

Tips

GTK 3.x (since 3.7)

GTK 3.x is supported. But the pre-compiled binaries, up to Linux 3.2, are still build with GTK 2.x. This is because GTK 3 implies in a big change for drawing applications, and most Tecgraf applications are heavily graphics dependent. The main change is that GDK does not have drawing primitives anymore, and all drawing is performed by Cairo. Cairo does not have support for XOR used by many drawing applications to perform a selection rubber band. Also Cairo could have a different behavior for some primitives. So while the applications are adapted to the new situation the pre-compiled files will remain with GTK 2.x

To build the driver with GTK 3.x support define USE_GTK3=Yes before calling make in the "iup/src" folder. Just the main library must be rebuilt. Also, if CD is used, the CD main library must be rebuilt with the same parameter.

Since Linux 3.13 the default is to use GTK 3.

Dependencies

GTK is in fact composed of several libraries. The GTK package contains the GDK library and depends on the ATK, Cairo, Glib and Pango libraries.

UTF-8

GTK uses UTF-8 as its charset for all displayed text, so IUP will automatically convert all strings to (SetAttribute) and from (GetAttribute) UTF-8. But the default is still to use the current locale. To use UTF-8 strings set the global attribute UTF8MODE to YES.

Windows

The GTK driver can be compiled and used in Windows, but it is not recommended since it is slower and much more memory consuming than the IUP native Windows driver.

When using DLLs in Windows, the "iup.dll" uses the Win32 driver. So applications that dynamically load IUP will always use the "iup.dll", for example Lua using require. To use the IUP GTK dll in Windows you must rename the "iupgtk.dll" to "iup.dll", so the GTK driver will be loaded instead of the Win32 driver.

After installing the GTK binaries, we recommend to change the default theme to the "MS-Windows" theme. Edit the "gtk\etc\gtk-2.0\gtkrc" file and change its contents to:

```
gtk-theme-name = "MS-Windows"
```

Ubuntu Unity

Since Ubuntu version 11 there is a new desktop called Unity. This desktop introduces some changes that affect all applications. Two of these changes directly affect IUP applications.

First the global menu forces all dialog menus to be displayed in the top of the desktop, like in MacOSX. This affected the size of the IUP dialog, it is fixed since IUP version 3.6. If you don't like the global menu you can remove it using:

```
sudo apt-get remove appmenu-gtk3 appmenu-gtk appmenu-qt
```

You can also control that using "export UBUNTU_MENUPROXY=0". There are other forms to control this feature see "How to Disable the AppMenu".

Second, the scrollbars are reduced to a very tiny line and handlers are displayed only when the mouse moves over the right or bottom side of the element. All the controls, except the **IupCanvas**, will work ok with the new scrollbar. But in **IupCanvas** the SCROLL_CB callback will receive only the IUP_SBPOSV and IUP_SBPOSH operations codes (fixed in IUP 3.11.1).

You can remove the new scrollbar at the **Synaptic Package Manager** or at the **Ubuntu Software Center** searching for "liboverlay-scrollbar" and removing the installed packages.

MacOSX-Quartz

The GTK driver also compiles in MacOSX with the new GTK port available at http://gtk-osx.sourceforge.net/ using Quartz. But the **IupGLCanvas** is not available yet. You must use the GDK base driver of the CD library. Some elements like **IupTree** are not 100% functional because of the gtk-osx implementation. The installation of gtk-osx is quite complex because there are no pre-compiled binaries. Also if the MacOSX theme is used, several controls have problems. We hope that this will improve in the future. Must define GTK_MAC before compiling to enable this build.

MacOSX-X11

So for now we are distributing binaries that use the X11 version of GTK 2.16. They were installed using Fink. Here is a simple guide to install fink so the pre-compiled binaries will work (tested in 10.5 and 10.6):

```
Download latest fink source: fink-0.29.21
tar -xvzf fink-0.29.21.tar.gz
cd fink-0.29.21
./bootstrap
 Use all default answers, except for the second question about 64bits:
 (10.5) => (1) Default (mostly 32bit)  [because gcc use 32bit as default]
 (10.6) => (2) 64bit-only              [because gcc use 64bit as default]
/sw/bin/pathsetup.sh
fink selfupdate-rsync
fink index -f
fink install gtk+2 gtk+2-dev
```

It will take some time to download and install everything, so have patience.

After that you should get something like this:

```
fink --version
  Package manager version: 0.29.10
  Distribution version: selfupdate-rsync Thu Apr 29 11:51:11 2010, 10.6, x86_64  (10.6)
  Distribution version: selfupdate-rsync Thu Apr 29 11:50:49 2010, 10.5, i386    (10.5)

fink list -i gtk+2
  Information about XXXX packages read in X seconds.
   i   gtk+2           2.16.6-3     The Gimp Toolkit
   i   gtk+2-dev       2.16.6-3     The Gimp Toolkit
   i   gtk+2-shlibs    2.16.6-3     The Gimp Toolkit
```

You can use MacPorts instead of Fink. The installation is very similar, but it seems simpler. Although we did not have the opportunity to test it, some users report that this work ok.

MacPorts have two GTK installations, one for X11 and one for Quartz. This is a lot simpler than trying to install the gtk-osx distribution. Just keep in mind that if using Quartz, OpenGL with IUP will not be available.

When building IUP or CD in MacOSX, define the following variables in the system before typing "make":

```
export USE_MACOS_OPENGL=Yes      # To use the OpenGL framework
export GTK_BASE=/sw              # For Fink
export GTK_BASE=/opt/local       # For MacPorts
export GTK_MAC=Yes               # For Quartz instead of X11
```

Attributes

Attributes are used to change or consult properties of elements. Each element has a set of attributes that affect it, and each attribute can work differently for each element. Depending on the element, its value can be computed or simply verified. Also it can be internally stored or not.

Attribute names are always upper case, lower case names will not work. But attribute values like "YES", "NO", "TOP", are case insensitive, so "Yes", "no", "top", and other variations will work.

If not defined their value can be inherited from the parent container.

Attributes Guide

Using

Attributes are a way to send and obtain information to and from elements. They are used by the application to communicate with the user interface system, on the other hand callbacks are used by the application to receive notifications from the system that the user or the system itself has interacted with the user interface of the application.

There are several functions to access attributes, see the documentation of the IupSetAttribute and IupGetAttribute for more options.

When an attribute is modified (**Set**) it is stored internally at the hash table of the control only if the control class allows it. If the value is NULL, the attribute will also be removed from the hash table and the default value will be used if there is one defined. Finally the attribute is updated for the children of the control if they do not have the attribute defined in their own hash table. Here is a pseudo-code:

```
IupSetAttribute(ih, name, value)
{
  if ih.SetClassAttribute(name, value)==store then
    ih.SetHashTableAttribute(name, value)
  endif

  if (ih.IsInheritable(name))
    -- NotifyChildren
    for each child of ih do
```

```
      if not child.GetHashTableAttribute(name) then
       child.SetClassAttribute(name, value)
       child.NotifyChildren(name, value)
      endif
    endfor
  endif
}
```

When an attribute is retrieved (**Get**) it will first be checked at the control class. If not defined then it checks in the hash table. If not defined it checks its parent hash table and so forth, until it reaches the dialog. And finally if still not defined then a default value is returned (the default value can also be NULL).

```
value = IupGetAttribute(ih, name)
{
  value = ih.GetClassAttribute(name)

  if not value then
    value = ih.GetHashTableAttribute(name)
  endif

  if not value and ih.isInheritable(name) then
    parent = ih.parent
    while (parent and not value)
      value = parent.GetHashTableAttribute(name)
      parent = parent.parent
    endwhile
  endif

  if not value then
    value = ih.GetDefaultAttribute(name)
  endif
}
```

Notice that the parent recursion is done only at the parent hash table, the parent control class is not consulted.

The control class can update or retrieve a value even if the control is not mapped. When the control is not mapped and its implementation can not process the attribute, then the attribute is simply stored in the hash table. After the element is mapped its attributes are re-processed to be updated in the native system and they can be removed from the hash table at that time.

All this flexibility turns the attribute system very complex with several nuances. If the attribute is checked before the control is mapped and just after, its value can be completely different. Depending on how the attribute is stored its inheritance can be completely ignored.

Attribute names are always upper case, lower case names will not work. But attribute values like "YES", "NO", "TOP", are case insensitive, so "Yes", "no", "top", and other variations will work.

Boolean attributes accept the values "1", "YES" or "ON" for **true**, and "0", "NO" or "OFF" for **false**. But they will return the value described in the documentation. You can also use **IupSetInt** with 1 for **true** and 0 for **false**. **IupGetInt** will return 1 for any of the **true** values, and 0 for any of the **false** values.

Floating point numbers when stored as strings use the application locale for decimal separator. Notice that by default C applications use the C locale, not the current system locale, in this case decimal separator is ".".

Combination of values in a single attribute is common, but there is no specific definitions on how they can be combined. Although all attributes that represent sizes using width and height adopt the "WxH" definition, for example "640x480". Position usually adopt "x,y" definition, range is usually "x1-x2" but can also be "x1:x2", so there are variations that for compatibility reasons were maintained. Cell specification is always "lin:col".

With **IupSetAttribute** you can also store application pointers that can be strings or not. This can be very useful, for instance, used inside **callbacks**. For example, by storing a C pointer of an application defined structure, the application can retrieve this pointer inside the callback through function **IupGetAttribute**. Therefore, even if the callbacks are global functions, the same callback can be used for several objects, even of different types.

Some controls, like **IupList**, **IupTree**, **IupTabs** and **IupMatrix**, have ids associated with some attributes so its value will affect only the respective id item in the control. For example: "TITLE3" will set the TITLE attribute for the item 3. To set that kind of attribute **IupSetAttribute** can be used, but **IupSetAttributeId** can also be used specially if the id is a variable.

There are attributes common to all the elements. In some cases, common attributes behave differently in different elements, but in such cases, there are comments in the documentation of the element explaining the different behavior.

In LED there is no quotation marks for attributes, names or values. In Lua attribute names can be lower case.

Inheritance

Elements included in other elements can inherit their attributes. There is an **inheritance** mechanism inside a given child tree.

This means, for example, that if you set the "MARGIN" attribute of a Vbox containing several other elements, including other Vboxes, all the elements depending on the attribute "MARGIN" will be affected, except for those who the "MARGIN" attribute is already defined.

Please note that not all attributes are inherited. As general rules the following attributes are **NON** inheritable always:

- Essential attributes like VALUE, TITLE, SIZE, RASTERSIZE, X and Y
- Id numbered attributes (like "1" or "MARK1:1")
- Handle names (like "CURSOR", "IMAGE" and "MENU")
- Pointers that are not strings (like WID)
- Read-only or write-only attributes
- Internal attributes that starts with "_IUP"

Inheritable attributes are stored in the hash table so the IupGet/SetAttribute logic can work, even if the control class store it internally. But when you change an attribute to NULL, then its value is removed from the hash table and the default value if any is passed to the native system.

When consulted the attribute is first checked at the control class. If not defined then it checks in the hash table. If not defined in its hash table, the attribute will be inherited from its parent's hash table and so forth, until it reaches the root child (usually the dialog). But if still then the attribute is not defined a default value for the element is returned (the default value can also be NULL).

When changed the attribute change is propagated to all children except for those who the attribute is already defined in the hash table.

But some attributes can be marked as **non inheritable** at the control class. (since 3.0)

Non inheritable attributes at the element are not propagated to its children. If an attribute is not marked as **non inheritable** at the element it is propagated as expected, but if marked as **non inheritable** at a child, that child will ignore the propagated value.

Since Vbox, Hbox, and other containers have only a few registered attributes, by default an unknown attribute is treated as inheritable, that's why it will be automatically propagated.

An example: the IMAGE attribute of a Label is **non inheritable**, so when checked at the Label it will return NULL if not defined, and the Label parent tree will not be consulted. If you change the IMAGE attribute at a Vbox that contains several Labels, the child Labels will not be affected.

Availability

Although attributes can be changed and retrieved at any time there are exceptions and some rules that must be followed according to the documentation of the attribute:

- **read only**: the attribute can not be changed. Ignored when set.
- **write only**: the attribute can not be retrieved. Normally used for action attributes. Returns NULL, or eventually some value set before the element was mapped.
- **creation only**: it will be used only when the element is mapped on the native system. So set it before the element is mapped. Ignored when set after the element is mapped.

IupLua

Each interface element is created as a Lua table, and its attributes are fields in this table**.** Some of these attributes are directly transferred to IUP, so that any changes made to them immediately reflect on the screen. However, not all attributes are transferred to IUP.

Control attributes, such as handle, which stores the handle of the IUP element, and parent, which stores the object immediately above in the class hierarchy, are not transferred. Attributes that receive strings or numbers as values are immediately transferred to IUP. Other values (such as functions or objects) are stored in IupLua and might receive special treatment.

For instance, a button can be created as follows (defining a title and the background color):

```
myButton = iup.button{title = "Ok", bgcolor = "0 255 0"}
```

Font color can be subsequently changed by modifying the value of attribute fgcolor:

```
myButton.fgcolor = "255 0 0"
```

Note that the attribute names in C and in IupLua are the same, but in IupLua they can be written in lower case.

In the creation of an element some parameters are required attributes (such as title in buttons). Their types are checked when the element is created. The required parameters are exactly the parameters that are necessary for the element to be created in C.

Some interface elements can contain one or more elements, as is the case of dialogs, lists and boxes. In such cases, the object's element list is put together as a vector, that is, the elements are placed one after the other, separated by commas. They can be accessed by indexing the object containing them, as can be seen in this example:

```
mybox = iup.hbox{bt1, bt2, bt3}
mybox[1].fgcolor = "255 0 0"         -- changes bt1 foreground color
mybox[2].fgcolor = caixa[1].fgcolor  -- changes bt2 foreground color
```

While the attributes receiving numbers or strings are directly transferred to IUP, attributes receiving other interface objects are not directly transferred, because IUP only accepts strings as a value. The method that transfers attributes to IUP verifies if the attribute value is a handle, that is, if it is an interface element. If the element already has a name, this name is passed to IUP. If not, a new name is created, associated to the element and passed to IUP as the value of the attribute being defined.

This policy is very useful for associating two interface elements, because you can abstract the fact that IUP uses a string to make associations and imagine the interface element itself is being used.

For attributes that contains two values combined the use of Lua can help splitting those values, for example:

```
w,h = string.match(ih.rastersize,"(%d*):(%d*)")
```

IupSetAttribute

Sets an interface element attribute. See also the Attributes Guide section.

Parameters/Return

```
void IupSetAttribute(Ihandle *ih, const char *name, const char *value); [in C]
void IupSetStrAttribute(Ihandle *ih, const char *name, const char *value); [in C]
iup.SetAttribute(ih: ihandle, name: string, value: any) [in Lua]

void IupSetAttributeId(Ihandle *ih, const char *name, int id, const char *value); [in C]
void IupSetStrAttributeId(Ihandle *ih, const char *name, int id, const char *value); [in C]
iup.SetAttributeId(ih: ihandle, name: string, id: number, value: any) [in Lua]

void IupSetAttributeId2(Ihandle* ih, const char* name, int lin, int col, const char* value); [in C]
void IupSetStrAttributeId2(Ihandle* ih, const char* name, int lin, int col, const char* value); [in C]
iup.SetAttributeId2(ih: ihandle, name: string, lin, col: number, value: any) [in Lua]
```

ih: Identifier of the interface element. If NULL will set in the global environment.
name: name of the attribute.
id, lin, col: used when the attribute has additional ids.
value: value of the attribute. If NULL (nil in Lua), the default value will be used.

Utility Functions

These functions can also be used to set attributes from the element:

```
void IupSetStrf  (Ihandle* ih, const char* name, const char* format, ...);
void IupSetInt   (Ihandle* ih, const char* name, int value);
void IupSetFloat (Ihandle* ih, const char* name, float value);
void IupSetDouble(Ihandle* ih, const char* name, double value);
void IupSetRGB   (Ihandle *ih, const char* name, unsigned char r, unsigned char g, unsigned char b);

void IupSetStrfId  (Ihandle *ih, const char* name, int id, const char* format, ...);
void IupSetIntId   (Ihandle* ih, const char* name, int id, int value);
void IupSetFloatId (Ihandle* ih, const char* name, int id, float value);
void IupSetDoubleId(Ihandle* ih, const char* name, int id, double value);
void IupSetRGBId   (Ihandle *ih, const char* name, int id, unsigned char r, unsigned char g, unsigned char b);

void IupSetStrfId2  (Ihandle* ih, const char* name, int lin, int col, const char* format, ...);
void IupSetIntId2   (Ihandle* ih, const char* name, int lin, int col, int value);
void IupSetFloatId2 (Ihandle* ih, const char* name, int lin, int col, float value);
void IupSetDoubleId2(Ihandle* ih, const char* name, int lin, int col, double value);
void IupSetRGBId2   (Ihandle *ih, const char* name, int lin, int col, unsigned char r, unsigned char g, unsigned char b);

[There is no equivalent in Lua]
```

IupSetStrf* functions (old **IupSetfAttribute**) uses the same format specification as the **sprintf** function in C. This function is very useful when several values must be combined into one string. When passing float values, it uses the format "%.9g" to maximize precision. When passing double values, it uses the format "%.18g" to maximize precision.

All the utility functions use the **IupSetStrAttribute*** functions internally.

Notes

See the Attributes Guide for more details.

IupSetAttribute can store only constant strings (like "Title", "30", etc) or application pointers. The given value is not duplicated as a string, only a reference is stored. Therefore, you can store application custom attributes, such as a context structure to be used in a callback.

IupSetStrAttribute (old **IupStoreAttribute**) can only store strings. The given string value will be duplicated internally.

Id based attributes are always non inheritable, so all **IupSet*Id** functions will not propagate the attribute to the children.

Examples

A very common mistake when using **IupSetAttribute** is to use local string arrays to set attributes. For ex:

```
char value[30];
sprintf(value, "CODE - %d", i);
IupSetAttribute(dlg, "BADEXAMPLE", value)  // WRONG  (value pointer will be internally stored,
                                           //         but its memory will be released at the end of this scope)
                                           // a common bad practice is to declare value as static
                                           // Use IupSetStrAttribute in this case
```

```
char *value = malloc(30);
sprintf(value, "%d", i);
IupSetAttribute(dlg, "EXAMPLE", value)     // correct  (but to avoid memory leaks you should free the pointer
                                                      after the dialog has been destroyed)
```

```
IupSetAttribute(dlg, "VISIBLE", "YES")     // correct (constant values still exists after this scope)
IupSetAttribute(text, "VALUE", "Hello!");
IupSetAttribute(indicator, "VALUE", "ON");
```

```
char attrib[30];
sprintf(attrib, "MY ITEM (%d)", i);
IupSetAttribute(dlg, attrib, "Test")       // correct (attribute names are always internally duplicated)
```

```
struct{
  int x;
  int y;
} myData;

IupSetAttribute(text, "myData", (char*)&myData);  // WRONG, will work only if myData is a global variable.
```

```
struct myData* mydata = malloc(sizeof(struct myData));
IupSetAttribute(dlg, "MYDATA", (char*)mydata);    // correct (unknown attributes will be stored as pointers)
```

Defines a radioâ€™s initial value:

```
Ihandle *portrait = IupToggle("Portrait" , NULL);
Ihandle *landscape = IupToggle("landscape" , NULL);
Ihandle *box = IupVbox(portrait, IupFill(),landscape, NULL);
Ihandle *mode = IupRadio(box);
IupSetHandle("landscape", landscape); /* associates a name to initialize the radio */
IupSetAttribute(mode, "VALUE", "landscape"); /* defines the radio's initial value */
```

See Also

IupGetAttribute, IupSetAttributes, IupGetAttributes, IupSetGlobal, IupGetGlobal

IupSetAttributes

Sets several attributes of an interface element.

Parameters/Return

```
Ihandle *IupSetAttributes(Ihandle *ih, const char *str); [in C]
iup.SetAttributes(ih: ihandle, str: string) -> ih: ihandle [in Lua]
```

ih: Identifier of the interface element.
str: string with the attributes in the format "v1=a1, v2=a2,..." where vi is the name of an attribute and ai is its value.

Returns: the same **ih**.

Examples

This function returns the same Ihandle it receives. This way, it is a lot easier to create dialogs in C. See also IupSetCallbacks.

```
dialog = IupSetAttributes(IupDialog(
    IupSetAttributes(IupHBox(
       canvas = IupSetAttributes(IupCanvas(NULL), "BORDER=NO, RASTERSIZE=100x100"),
       NULL), "MARGIN=10x10"),
    "TITLE=Test");
```

Creates a list with country names and defines Japan as the selected option.

```
Ihandle *list = IupList (NULL);
IupSetAttributes(list,"VALUE=3,1=Brazil,2=USA,3=Japan,4=France");
```

See Also

IupGetAttribute, IupSetAttribute, IupGetAttributes, IupSetAtt

IupResetAttribute (Since 3.2)

Removes an attribute from the hash table of the element, and its children if the attribute is inheritable. It is useful to reset the state of inheritable attributes in a tree of elements.

Parameters/Return

```
void IupResetAttribute(Ihandle *ih, const char *name); [in C]
iup.ResetAttribute(ih: ihandle, name: string) [in Lua]
```

ih: Identifier of the interface element. If NULL will set in the global environment.
name: name of the attribute.

See Also

IupGetAttribute, IupSetAttribute

IupSetAtt

Sets several attributes of an interface element and optionally sets its name.

Parameters/Return

```
Ihandle* IupSetAtt(const char* handle_name, Ihandle* ih, const char* name, ...); [in C]
```

handle_name: optional handle name. **IupSetHandle** will be called internally. can be NULL**.**
ih: Identifier of the interface element.
name: name of the first attribute.
...: after **name** a **value** must be set, then a sequence of name and value pairs can follow until a NULL name is found. It must be a constant string because **IupSetAttribute** will be used internally.

Returns: **ih**

Examples

This function returns the same Ihandle it receives. This way, it is a lot easier to create dialogs in C. See also IupSetCallbacks.

```
dialog = IupSetAtt("MainDialog", IupDialog(
    IupSetAtt(NULL, IupHBox(
       IupSetAtt("MainCanvas", IupCanvas(NULL), "BORDER", "NO", "RASTERSIZE", "100x100", NULL),
       NULL), "MARGIN", "10x10", NULL),
    "TITLE", "Test", NULL);
```

Creates a list with country names and defines Japan as the selected option.

```
Ihandle *list = IupList(NULL);
IupSetAtt(NULL, list, "VALUE", "3", "1", "Brazil", "2", "USA", "3", "Japan", "4", "France", NULL);
```

See Also

IupGetAttribute, IupSetAttribute, IupGetAttributes, IupSetAttributes

IupSetAttributeHandle

Instead of using **IupSetHandle** and **IupSetAttribute** with a new creative name, this function automatically creates a non conflict name and associates the name with the attribute.

It is very useful for associating images and menus.

Parameters/Return

```
void IupSetAttributeHandle(Ihandle *ih, const char *name, Ihandle *ih_named); [in C]
[There is no equivalent in Lua]
```

ih: identifier of the interface element.
name: name of the attribute.
ih_named: element to associate using a name

The function will not check for inheritance since all the attributes that associate handles are not inheritable.

Notes

This work is automatically done in Lua when an attribute that is an element name is set to an element handle. In other words, in Lua you can set a string or a handle as the attribute value, when a handle is used a name is automatically created just as the **IupSetAttributeHandle**.

See Also

IupGetAttributeHandle, IupSetAttribute, IupSetAttributes, IupSetHandle

IupGetAttribute

Returns the name of an interface element attribute. See also the Attributes Guide section.

Parameters/Return

```
char *IupGetAttribute(Ihandle *ih, const char *name); [in C]
iup.GetAttribute(ih: ihandle, name: string) -> value: string, ihandle or userdata [in Lua]

char *IupGetAttributeId(Ihandle *ih, const char *name, int id); [in C]
iup.GetAttributeId(ih: ihandle, name: string, id: number) -> value: string, ihandle or userdata [in Lua]

char* IupGetAttributeId2(Ihandle* ih, const char* name, int lin, int col); [in C]
iup.GetAttributeId2(ih: ihandle, name: string, lin, col: number) -> value: string, ihandle or userdata [in Lua]
```

ih: Identifier of the interface element. If NULL will retrieve from the global environment.
name: name of the attribute.
id, lin, col: used when the attribute has additional ids.

Returns: the attribute value or NULL (nil in Lua) if the attribute is not set or does not exist.

Utility Functions

These functions can also be used to get attributes from the element:

```
int    IupGetInt   (Ihandle* ih, const char* name);
int    IupGetIntInt(Ihandle* ih, const char* name, int *i1, int *i2);
int    IupGetInt2  (Ihandle* ih, const char* name);
float  IupGetFloat (Ihandle* ih, const char* name);
double IupGetDouble(Ihandle* ih, const char* name);
void   IupGetRGB   (Ihandle *ih, const char* name, unsigned char *r, unsigned char *g, unsigned char *b);

int    IupGetIntId   (Ihandle* ih, const char* name, int id);
float  IupGetFloatId (Ihandle* ih, const char* name, int id);
```

```
double IupGetDoubleId(Ihandle* ih, const char* name, int id);
void   IupGetRGBId   (Ihandle *ih, const char* name, int id, unsigned char *r, unsigned char *g, unsigned char *b);

int    IupGetIntId2   (Ihandle* ih, const char* name, int lin, int col);
float  IupGetFloatId2 (Ihandle* ih, const char* name, int lin, int col);
double IupGetDoubleId2(Ihandle* ih, const char* name, int lin, int col);
void   IupGetRGBId2   (Ihandle *ih, const char* name, int lin, int col, unsigned char *r, unsigned char *g, unsigned char *b);

[There is no equivalent in Lua]
```

IupGetIntInt retrieves two integers separated by 'x', ':' or ',' and returns the number of returned values (0, 1 or 2). **IupGetInt2** returns just the second value.

Notes

See the Attributes Guide for more details.

The returned value is not necessarily the same pointer used by the application to define the attribute value. The pointers of internal IUP attributes returned by **IupGetAttribute** should **never** be freed or changed, except when it is a custom application pointer that was stored using **IupSetAttribute** and allocated by the application.

The returned pointer can be used safely even if **IupGetGlobal** or **IupGetAttribute** are called several times. But not too many times, because it is an internal buffer and after IUP may reuse it after around 50 calls.

IupLua

In IupLua, only known internal pointer attributes are returned as **user data** or as an ihandle, all other attributes are returned as strings. To access attribute data always as **user data** use iup.GetAttributeData:

```
iup.GetAttributeData(ih: ihandle) -> value: userdata [in Lua]
```

Examples

Browse for Example Files

See Also

IupSetAttribute, IupSetAttributes, IupGetHandle, IupSetGlobal, IupGetGlobal

IupGetAllAttributes (Since 3.0)

Returns the names of all attributes of an element that are set in its internal hash table only.

Parameters/Return

```
int IupGetAllAttributes(Ihandle* ih, char** names, int max_n); [in C]
iup.GetAllAttributes(ih: ihandle[, max_n: number]) -> (names: table, n: number) [in Lua]
```

ih: identifier of the interface element.
names: table receiving the names. Only the list of names need to be allocated. Each name will point to an internal string.
max_n: maximum number of names the table can receive. Can be omitted in Lua.

Returns: the actual number of names loaded to the table. If names==NULL or max_n==0 then returns the maximum number of names.

See Also

IupGetAttribute, IupSetAttribute, IupSetAttributes

IupGetAttributes

Returns all attributes of a given element that are set in the internal hash table. The known attributes that are pointers (not strings) are returned as integers.

The internal attributes are not returned (attributes prefixed with "_IUP").

Before calling this function the application must ensure that there is no pointer attributes set for that element, although all known pointers attributes are handled.

This function should be avoided. Use **IupGetAllAttributes** instead.

Parameters/Return

```
char* IupGetAttributes (Ihandle *ih); [in C]
iup.GetAttributes(ih: ihandle) -> (ret: string) [in Lua]
```

ih: Identifier of the interface element.

Returns: a string with all attributes in the format: "NAME=VALUE, NAME="VALUE", ...".

See Also

IupGetAttribute, IupGetAllAttributes, IupSetAttribute, IupSetAttributes

IupGetAttributeHandle

Instead of using IupGetAttribute and IupGetHandle, this function directly returns the associated handle.

Parameters/Return

```
Ihandle* IupGetAttributeHandle(Ihandle *ih, const char *name); [in C]
[There is no equivalent in Lua]
```

ih: identifier of the interface element.
name: name of the attribute.

Returns: the element with the associated name. The function will not check for inheritance since all the attributes that associate handles are not inheritable.

See Also

IupSetAttributeHandle, IupSetAttribute, IupSetAttributes, IupSetHandle

IupSetGlobal

Sets an attribute in the global environment. If the driver process the attribute then it will not be stored internally.

Parameters/Return

```
void IupSetGlobal(const char *name, const char *value); [in C]
void IupSetStrGlobal(const char *name, const char *value); [in C]
iup.SetGlobal(name: string, value: string) [in Lua]
```

name: name of the attribute.
value: value of the attribute. If it equals NULL (nil in IupLua), the attribute will be removed.

Notes

IupSetGlobal can store only constant strings (like "Title", "30", etc) or application pointers. The given value is not duplicated as a string, only a reference is stored. Therefore, you can store application custom attributes, such as a context structure to be used in a callback.

IupSetStrGlobal (old **IupStoreGlobal**) can only store strings. The given string value will be duplicated internally.

IupSetAttribute functions can also be used to set global attributes, just set the element to NULL.

See Also

IupGetGlobal, Global Attributes

IupGetGlobal

Returns an attribute value from the global environment. The value can be returned from the driver or from the internal storage.

Parameters/Return

```
char *IupGetGlobal(const char *name); [in C]
iup.GetGlobal(name: string) -> value: string [in Lua]
```

name: name of the attribute.

Returns: the attribute value. If the attribute does not exist, NULL (nil in Lua) is returned.

Notes

This function's return value is not necessarily the same one used by the application to set the attribute's value.

The returned value is not necessarily the same pointer used by the application to define the attribute value. The pointers of internal IUP attributes returned by **IupGetGlobal** should **never** be freed or changed, except when it is a custom application pointer that was stored using **IupSetGlobal** and allocated by the application.

The returned pointer can be used safely even if **IupGetGlobal** or **IupGetAttribute** are called several times. But not too many times, because it is an internal buffer and after IUP may reuse it after around 50 calls.

IupGetAttribute can also be used to get global attributes, just set the element to NULL.

See Also

IupSetGlobal, Global Attributes

IupStringCompare (since 3.17)

Utility function to compare strings lexicographically. Used internally in **IupMatrixEx** when sorting, but available in the main library.

This means that numbers and text in the string are sorted separately (for ex: A1 A2 A11 A30 B1). Also natural alphabetic order is used: 123...aAáÁ...bBcC... The comparison will work only for Latin-1 characters, even if UTF8MODE is Yes.

Parameters/Return

```
void IupStringCompare(const char* str1, const char* str2, int casesensitive, int lexicographic); [in C]
iup.StringCompare(str1, str2: string[, casesensitive, lexicographic: number]) [in Lua]
```

str1 and str2: strings to be compared.
casesensitive: flag to enable case sensitive compare. Can be 0 (disable) or 1 (enable). In Lua the default value is 0.
lexicographic: flag to enable lexicographic compare. Can be 0 (disable) or 1 (enable). When disabled the compare will only return if the strings are equal (0) or different (1). In Lua the default value is 1.

Returns: 0 if str1 == str2, -1 if str1<str2, 1 if str1 > str2 (same return values of the **strcmp** function).

Notes

The Alphanum Algorithm is discussed at http://www.DaveKoelle.com/alphanum.html.

This implementation is Copyright (c) 2008 Dirk Jagdmann <doj@cubic.org>. It is a cleanroom implementation of the algorithm and not derived by other's works. In contrast to the versions written by Dave Koelle this source code is distributed with the libpng/zlib license.

The IUP implementation is based on the "alphanum.hpp" code downloaded from the Dave Koelle page and implemented by Dirk Jagdmann. It was modified to the C language and simplified to IUP needs.

See Also

IupMatrixEx

ACTIVE

Activates or inhibits user interaction.

Value

"YES" (active), "NO" (inactive).

Default: "YES"

Notes

An interface element is only active if its native parent is also active.

ACTIVE can also be set for controls that do not have user interaction because they may have a visual feedback to indicate the inactive state.

In GTK and Motif the inactive dialogs will still be able to move, resize and change their Z-order. Although their contents will be inactive, keyboard will be disabled, and they can not be closed from the close box.

Affects

All controls that have visual representation.

BGCOLOR

Element's background color.

Value

The RGB components.

Values should be between 0 and 255, separated by a blank space. For example "255 0 128", red=255 blue=0 green=128.

Default: It is the value of the DLGBGCOLOR global attribute. On some controls if not defined will inherit the background of the native parent.

Hexadecimal notation in the format "#RRGGBB" is also accepted in all color attributes. For example, "255 0 128" can also be written as "#FF0080".

Affects

All controls that have visual representation, but with some restrictions.

Several controls have transparent parts that are not affected by the BGCOLOR.

See also the screenshots of the sample.c results with normal background, changing the dialog BACKGROUND, the dialog BGCOLOR and the children BGCOLOR.

See Also

FGCOLOR, DLGBGCOLOR

FGCOLOR

Element's foreground color. Usually it is the color of the associated text.

Value

The RGB components. Values should be between 0 and 255, separated by a blank space.

Default: "0 0 0".

Hexadecimal notation in the format "#RRGGBB" is also accepted in all color attributes. For example, "255 0 128" can also be written as "#FF0080".

Affects

All controls that have visual representation.

See Also

BGCOLOR

FONT

Character font of the text shown in the element. Although it is an inheritable attribute, it is defined only on elements that have a native representation, but other elements are affected because it affects the SIZE attribute.

Value

Font description containing typeface, style and size. Default: the global attribute DEFAULTFONT.

The common format definition is similar to the the Pango library Font Description, used by GTK+. It is defined as having 3 parts: "<font face>, <font styles> <font size>".

Font face is the font face name, and can be any name. Although only names recognized by the system will be actually used. The names Helvetica, Courier and Times are always accepted in all systems.

The supported font style is a combination of: **Bold**, **Italic**, **Underline** and **Strikeout**. The Pango format include many other definitions not supported by the common format, they are supported only by the GTK driver. Unsupported values are simply ignored. The names must be in the same case describe here.

Font size is in points (1/72 inch) or in pixels (using negative values).

Returned values will be the same value when changing the attribute, except for the old font names that will be converted to the new common format definition.

Windows

The DEFAULTFONT is retrieved from the System Settings (see below), if this failed then "Tahoma, 10" for Windows XP, or "Segoe UI, 9" since Windows Vista, is assumed.

The native handle can be obtained using the "**HFONT**" attribute.

In "Control Panel", open the "Display Properties" then click on "Advanced" and select "Message Box" and change its Font to affect the default font for applications. In Vista go to "Window Color and Appearance", then "Open Classic Appearance", then Advanced.

Motif

The DEFAULTFONT is retrieved from the user resource file (see below), if failed then "Fixed, 11" is assumed.

The X-Windows Logical Font Description format (XLFD) is also supported.

The native handle can be obtained using the "**XMFONTLIST**" and "**XFONTSTRUCT**" attributes. The selected X Logical Font Description string can be obtained from the attribute "**XLFD**".

You can use the **xfontsel** program to obtain a string in the X-Windows Logical Font Description format (XLFD). Noticed that the first size entry of the X-Windows font string format (**pxlsz**) is in pixels and the next one (**ptSz**) is in deci-points (multiply the value in points by 10).

Be aware that the resource files ".Xdefaults" and "Iup" in the user home folder can affect the default font and many others visual appearance features in Motif.

GTK

The DEFAULTFONT is retrieved from the style defined by GNOME (see below), if failed "Sans, 10" is assumed.

The X-Windows Logical Font Description format (XLFD) is also supported.

The native handle can be obtained using the "**PANGOFONTDESC**" attribute.

Font face can be a list of fonts face names in GTK. For example "Arial,Helvetica, 12". Not accepted in the other drivers.

Style can also be a combination of: Small-Caps, [Ultra-Light|Light|Medium|Semi-Bold|Bold|Ultra-Bold|Heavy], [Ultra-Condensed|Extra-Condensed|Condensed|Semi-Condensed|Semi-Expanded|Expanded|Extra-Expanded|Ultra-Expanded]. Those values can be used only when the string is a full Pango compliant font, i.e. no underline, no strikeout and size>0.

In GNOME, go to the "Appearance Preferences" tool, then in the Fonts tab change the Applications Font to affect the default font.

Examples:

```
"Times, Bold 18"
"Arial, 24" (no style)
"Courier New, Italic Underline -30" (size in pixels)
```

Affects

All elements, since the SIZE attribute depends on the FONT attribute, except for menus.

Notes

When FONT is changed and SIZE is set, then RASTERSIZE is also updated.

Since font face names are not a standard between Windows, Motif and GTK, a few names are specially handled to improve application portability. If you want to use names that work for all systems we recommend using: Courier, Times and Helvetica (same as Motif). Those names always have a native system name equivalent. If you use those names IUP will automatically map to the native system equivalent. See the table below:

Recommended/Motif	Windows	GTK	Description
Helvetica	Arial	Sans	without serif, variable spacing
Courier	Courier New	Monospace	with serif, fixed spacing
Times	Times New Roman	Serif	with serif, variable spacing

Auxiliary Attributes

They will change the FONT attribute, and depends on it. They are used only to set partial FONT parameters of style and size. To do that the FONT attribute is parsed, changed and updated to the new value in the common format definition. This means that if the attribute was set in X-Windows format or in the old Windows and IUP formats, the previous value will be replaced by a new value in the common format definition. Pango additional styles will also be removed.

FONTSTYLE (non inheritable)

Replaces or returns the style of the current FONT attribute. Since font styles are case sensitive, this attribute is also case sensitive.

FONTSIZE (non inheritable)

Replaces or returns the size of the current FONT attribute.

FONTFACE (non inheritable)

Replaces or returns the face name of the current FONT attribute.

CHARSIZE (read-only, non inheritable)

Returns the average character size of the current FONT attribute. This is the factor used by the SIZE attribute to convert its units to pixels.

FOUNDRY [Motif Only] (non inheritable)

Allows to select a foundry for the FONT being selected. Must be set before setting the FONT attribute.

Encoding

The number that represents each character is dependent on the encoding used. For example, in ASCII encoding the letter A has code 65, but codes above 128 can be very different according to the encoding. An encoding is defined by a charset.

IUP uses the default locale in ANSI-C. This means that it does not adopts a specific charset, and so the results can be different if the developer charset is different than the run time charset where the user is running the application. For example, if the developer is using a charset, and its user is also using the same encoding, then everything will work fine without the need of text encoding conversions. The advantage is that any charset can be used, and localization is usually done in that way.

Since version 3.9, IUP supports also the UTF-8 (ISO10646-1) encoding in the GTK and Windows drivers. To specify a string in UTF-8 encoding set the global attribute "UTF8MODE" to "Yes".

ISO8859-1 and Windows-1252 Displayable Characters

The Latin-1 charset (ISO8859-1) defines an encoding for all of the characters used in Western languages. It is the most common encoding, besides UTF-8.

The first half of Latin-1 is standard ASCII (128 characters), while the second half (with the highest bit set) contains accented characters needed for Western languages other than English. In Windows, the ISO8859-1 charset was changed, and some control characters were replaced to include more display characters, this new charset is named Windows-1252 (these characters are marked in the table below with thick borders).

	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
32	***33***	***34***	***35***	***36***	***37***	***38***	***39***	***40***	***41***	***42***	***43***	***44***	***45***	***46***	***47***
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?

48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	} 125	~ 126	127
€ 128	129	‚ 130	ƒ 131	„ 132	… 133	† 134	‡ 135	ˆ 136	‰ 137	Š 138	‹ 139	Œ 140	141	Ž 142	143
144	‘ 145	’ 146	“ 147	” 148	• 149	– 150	— 151	˜ 152	™ 153	š 154	› 155	œ 156	157	ž 158	Ÿ 159
160	¡ 161	¢ 162	£ 163	¤ 164	¥ 165	¦ 166	§ 167	¨ 168	© 169	ª 170	« 171	¬ 172	173	® 174	¯ 175
° 176	± 177	² 178	³ 179	´ 180	µ 181	¶ 182	· 183	¸ 184	¹ 185	º 186	» 187	¼ 188	½ 189	¾ 190	¿ 191
À 192	Á 193	Â 194	Ã 195	Ä 196	Å 197	Æ 198	Ç 199	È 200	É 201	Ê 202	Ë 203	Ì 204	Í 205	Î 206	Ï 207
Ð 208	Ñ 209	Ò 210	Ó 211	Ô 212	Õ 213	Ö 214	× 215	Ø 216	Ù 217	Ú 218	Û 219	Ü 220	Ý 221	Þ 222	ß 223
à 224	á 225	â 226	ã 227	ä 228	å 229	æ 230	ç 231	è 232	é 233	ê 234	ë 235	ì 236	í 237	î 238	ï 239
ð 240	ñ 241	ò 242	ó 243	ô 244	õ 245	ö 246	÷ 247	ø 248	ù 249	ú 250	û 251	ü 252	ý 253	þ 254	ÿ 255

	Punctuation and Symbols
	Numbers
	Letters
	Accented

Adapted from http://en.wikipedia.org/wiki/Windows-1252.

UTF-8

"Universal character set Transformation Format - 8 bits" is part of the Unicode standard that is used in most modern Web applications and it is becoming widely used in desktop applications too.

It allows the application to use a regular "char*" for strings, but it is a variable width encoding, meaning that a single character may have up to four bytes in sequence. And the code "\0" is still used as a string terminator (NULL). So all the regular **strstr**, **strcmp**, **strlen**, **strcpy** and **strcat** functions will work normally, except **strchr** because it will search only for 1 byte characters. Notice that **strlen** will return the number of bytes, not the number of multi-byte charcters. And **strcmp** will compare byte encodings.

The first 128 characters of Unicode, which correspond one-to-one with ASCII, are encoded using a single octet with the same binary value as ASCII, making valid ASCII text valid UTF-8-encoded Unicode as well. If the highest bit is 1 then one to three more bytes will follow to the define the actual character encoding. The number of bytes following is determined by the number of bits set to 1 after the highest bit.

The next 1920 characters need two bytes to encode. This covers the remainder of almost all Latin-derived alphabets, and also Greek, Cyrillic, Coptic, Armenian, Hebrew, Arabic, Syriac and Tāna alphabets, as well as Combining Diacritical Marks. Three bytes are needed for characters in the rest of the Basic Multilingual Plane (which contains virtually all characters in common use[11]). Four bytes are needed for characters in the other planes of Unicode, which include less common CJK characters and various historic scripts and mathematical symbols.

The bytes 0xFE and 0xFF do not appear, so a valid UTF-8 string can cannot be confused with an UTF-16 sequence.

The second half (128-255) of the Latin-1 charset characters found in the previous table, are called "Latin-1 Supplement" in the Unicode standard. They all have two bytes, except some of the additional Windows 1252 characters. And they have the following encoding in UTF-8 (codes in hexadecimal):

€ E2 82 AC		‚ E2 80 9A	ƒ C6 92	„ E2 80 9E	… E2 80 A6	† E2 80 A0	‡ E2 80 A1	ˆ CB 86	‰ E2 80 B0	Š C5 A0	‹ E2 80 B9	Œ C5 92		Ž C5 BD	
	‘ E2 80 98	’ E2 80 99	“ E2 80 9C	” E2 80 9D	• E2 80 A2	– E2 80 93	— E2 80 94	˜ CB 9C	™ E2 84 A2	š C5 A1	› E2 80 BA	œ C5 93		ž C5 BE	Ÿ C5 B8
C2 A0	¡ C2 A1	¢ C2 A2	£ C2 A3	¤ C2 A4	¥ C2 A5	¦ C2 A6	§ C2 A7	¨ C2 A8	© C2 A9	ª C2 AA	« C2 AB	¬ C2 AC	C2 AD	® C2 AE	¯ C2 AF
° C2 B0	± C2 B1	² C2 B2	³ C2 B3	´ C2 B4	µ C2 B5	¶ C2 B6	· C2 B7	¸ C2 B8	¹ C2 B9	º C2 BA	» C2 BB	¼ C2 BC	½ C2 BD	¾ C2 BE	¿ C2 BF
À C3 80	Á C3 81	Â C3 82	Ã C3 83	Ä C3 84	Å C3 85	Æ C3 86	Ç C3 87	È C3 88	É C3 89	Ê C3 8A	Ë C3 8B	Ì C3 8C	Í C3 8D	Î C3 8E	Ï C3 8F
Ð C3 90	Ñ C3 91	Ò C3 92	Ó C3 93	Ô C3 94	Õ C3 95	Ö C3 96	× C3 97	Ø C3 98	Ù C3 99	Ú C3 9A	Û C3 9B	Ü C3 9C	Ý C3 9D	Þ C3 9E	ß C3 9F
à C3 A0	á C3 A1	â C3 A2	ã C3 A3	ä C3 A4	å C3 A5	æ C3 A6	ç C3 A7	è C3 A8	é C3 A9	ê C3 AA	ë C3 AB	ì C3 AC	í C3 AD	î C3 AE	ï C3 AF
ð C3 B0	ñ C3 B1	ò C3 B2	ó C3 B3	ô C3 B4	õ C3 B5	ö C3 B6	÷ C3 B7	ø C3 B8	ù C3 B9	ú C3 BA	û C3 BB	ü C3 BC	ý C3 BD	þ C3 BE	ÿ C3 BF

Adapted from http://en.wikipedia.org/wiki/UTF-8

VISIBLE

Shows or hides the element.

Value

"YES" (visible), "NO" (hidden).

Default: "YES"

Notes

An interface element is only visible if its native parent is also visible.

Affects

All controls that have visual representation, except menus.

CLIENTSIZE (read-only*) (non inheritable) (since 3.0)

Returns the client area size of a container. It is the space available for positioning and sizing children, see the Layout Guide. It is the container **Current** size excluding the decorations (if any).

Value

"*width*x*height*", where *width* and *height* are integer values corresponding to the horizontal and vertical size, respectively, in pixels.

Affects

All elements that are containers, except menus.

Notes

For **IupHbox**, **IupVbox** and **IupGridBox** it consider the MARGIN attribute as a decoration.

For **IupDialog**, **IupBackgroundBox**, **IupScrollBox** and **IupGLCanvasBox** is only available after the control is mapped.

For **IupDialog** is NOT read-only, and it will re-define RASTERSIZE by adding the decorations to the actual Client size. (Since 3.3)

For **IupSplit** returns the total area available for the two children.

See Also

SIZE, RASTERSIZE, CLIENTOFFSET

CLIENTOFFSET (read-only) (non inheritable) (since 3.3)

Returns the container native offset to the **Client** area, see the Layout Guide. Useful for **IupFrame**, **IupTabs** and **IupDialog** that have decorations. Can also be consulted in other containers, it will simply return "0x0".

This attribute can be used in conjunction with the POSITION attribute of a child so the coordinates of a child relative to the native parent can be obtained.

Value

"*dx*x*dy*", where *dx* and *dy* are integer values corresponding to the horizontal and vertical offsets, respectively, in pixels.

Affects

All elements that are containers, except menus.

Notes

For **IupDialog** is only available after the control is mapped.

In GTK and Motif, for the **IupDialog**, the dy value is negative when there is a menu. This occurs because in those systems the menu is placed inside the Client Area and all children must be placed below the menu.

In Windows, for the **IupFrame**, the value is always "0x0" the position of the child is still relative to the top-left corner of the frame. This is automatically compensated in calculation of the POSITION attribute.

See Also

SIZE, RASTERSIZE, CLIENTSIZE, POSITION

EXPAND (non inheritable*)

Allows the element to expand, fulfilling empty spaces inside its container.

It is a non inheritable attribute, but a container inherit its parents EXPAND attribute. In other words, although EXPAND is non inheritable, it is inheritable for containers. So if you set it at a container it will not affect its children, except for those who are containers.

The expansion is done equally for all expandable elements in the same container.

For a container, the actual EXPAND value will be always a **combination** of its own value and the value of its **children**. In the sense that a container can only expand if its children can expand too in the same direction.

The HORIZONTALFREE and VERTICALFREE values will not behave as normal expansion. These values will NOT affect the expansion of the container when set at its children, the children will simply expand to the available free space at the container. (Since 3.11)

See the Layout Guide for more details on sizes.

Value

"YES" (both directions), "HORIZONTAL", "VERTICAL", "HORIZONTALFREE", "VERTICALFREE" or "NO".

Default: "NO". For containers the default is "YES".

Affects

All elements, except menus.

MAXSIZE (non inheritable) (since 3.0)

Specifies the element maximum size in pixels during the layout process.

See the Layout Guide for more details on sizes.

Value

"*width*x*height*", where *width* and *height* are integer values corresponding to the horizontal and vertical size, respectively, in pixels.

You can also set only one of the parameters by removing the other one and maintaining the separator "x", but this is equivalent of setting the other value to 0. For example: "x40" (height only = "0x40") or "40x" (width only = "40x0").

Affects

All, except menus.

Notes

The limits are applied during the layout computation. It will limit the Natural size and the Current size.

If the element can be expanded, then its empty space will NOT be occupied by other controls although its size will be limited.

In the **IupDialog** will also limit the interactive resize of the dialog.

See the Layout Guide for mode details on sizes.

See Also

RASTERSIZE, MINSIZE

MINSIZE (non inheritable) (since 3.0)

Specifies the element minimum size in pixels during the layout process.

See the Layout Guide for more details on sizes.

Value

"*width*x*height*", where *width* and *height* are integer values corresponding to the horizontal and vertical size, respectively, in pixels.

You can also set only one of the parameters by removing the other one and maintaining the separator "x", but this is equivalent of setting the other value to 0. For example: "x40" (height only = "0x40") or "40x" (width only = "40x0").

Affects

All, except menus.

Notes

The limits are applied during the layout computation. It will limit the Natural size and the Current size.

If the element can be expanded, then its empty space will NOT be occupied by other controls although its size will be limited.

In the **IupDialog** will also limit the interactive resize of the dialog.

See the Layout Guide for mode details on sizes.

See Also

RASTERSIZE, MAXSIZE

NATURALSIZE (non inheritable, read-only)

Returns the element last computed **Natural** size in pixels.

See the Layout Guide for more details on sizes.

Value

"*width*x*height*", where *width* and *height* are integer values corresponding to the horizontal and vertical size, respectively, in pixels.

See Also

SIZE, RASTERSIZE

RASTERSIZE (non inheritable)

Specifies the element **User** size, and returns the **Current** size, in pixels.

See the Layout Guide for more details on sizes.

Value

"*width*x*height*", where *width* and *height* are integer values corresponding to the horizontal and vertical size, respectively, in pixels.

You can also set only one of the parameters by removing the other one and maintaining the separator "x", but this is equivalent of setting the other value to 0. For example: "x40" (height only = "0x40") or "40x" (width only = "40x0").

When this attribute is consulted the **Current** size of the control is returned.

Affects

All, except menus.

Notes

When this attribute is set, it resets the SIZE attribute. So changes to the FONT attribute will not affect the **User** size of the element.

To obtain the last computed **Natural** size of the control in pixels, use the read-only attribute NATURALSIZE. (Since 3.6)

To obtain the **User** size of the element in pixels after it is mapped, use the attribute **USERSIZE**. (Since 3.12)

A **User** size of "0x0" can be set, it can also be set using NULL. If both values are 0 then NULL is returned.

If you wish to use the **User** size only as an initial size, change this attribute to NULL after the control is mapped, the returned size in **IupGetAttribute** will still be the **Current** size.

The element is NOT immediately repositioned. Call **IupRefresh** to update the dialog layout.

IupMap also updates the dialog layout even if it is already mapped, so calling it or calling **IupShow**, **IupShowXY** or **IupPopup** (they all call **IupMap**) will also update the dialog layout.

See the Layout Guide for mode details on sizes.

See Also

SIZE, FONT

SIZE (non inheritable)

Specifies the element **User** size, and returns the **Current** size, in units proportional to the size of a character.

See the Layout Guide for more details on sizes.

Value

"*width*x*height*", where width and height are integer values corresponding to the horizontal and vertical size, respectively, in characters fraction unit (see Notes below).

You can also set only one of the parameters by removing the other one and maintaining the separator "x", but this is equivalent of setting the other value to 0. For example: "x40" (height only = "0x40") or "40x" (width only = "40x0").

When this attribute is consulted the **Current** size of the control is returned.

Notes

The size units observes the following heuristics:

- Width in 1/4's of the average width of a character for the current **FONT** of each control.
- Height in 1/8's of the average height of a character for the current **FONT** of each control.

So, a SIZE="4x8" means 1 character width and 1 character height.

Notice that this is the average character size, the space occupied by a specific string is always different than its number of character times its average character size, except when using a monospaced font like Courier. Usually for common strings this size is smaller than the actual size, so it is a good practice to leave more room than expected if you use the SIZE attribute. For smaller font sizes this difference is more noticeable than for larger font sizes.

When this attribute is changed, the RASTERSIZE attribute is automatically updated.

SIZE depends on FONT, so when **FONT** is changed and **SIZE** is set, then **RASTERSIZE** is also updated.

The average character size of the current **FONT** can be obtained from the CHARSIZE attribute.

To obtain the last computed **Natural** size of the element in pixels, use the read-only attribute NATURALSIZE. (Since 3.6)

To obtain the **User** size of the element in pixels after it is mapped, use the attribute **USERSIZE**. (Since 3.12)

A **User** size of "0x0" can be set, it can also be set using NULL. If both values are 0 then NULL is returned.

If you wish to use the **User** size only as an initial size, change this attribute to NULL after the control is mapped, the returned size in **IupGetAttribute** will still be the **Current** size.

The element is NOT immediately repositioned. Call **IupRefresh** to update the dialog layout.

IupMap also updates the dialog layout even if it is already mapped, so calling it or calling **IupShow**, **IupShowXY** or **IupPopup** (they all call **IupMap**) will also update the dialog layout.

See the Layout Guide for mode details on sizes.

Affects

All, except menus.

See Also

FONT, RASTERSIZE, IupRefresh

FLOATING (non inheritable) (since 3.0)

If an element has FLOATING=YES then its size and position will be ignored by the layout processing in **IupVbox**, **IupHbox** and **IupZbox**. But the element size and position will still be updated in the native system allowing the usage of SIZE/RASTERSIZE and POSITION to manually position and size the element. And must ensure that the element will be on top of other using ZORDER, if there is overlap.

This is useful when you do not want that an invisible element to be computed in the box size.

If the value IGNORE is used then it will behave as YES, but also it will not update the the size and position in the native system. (since 3.3)

Value

"YES", "IGNORE" (since 3.3) or "NO".

Default: "NO".

Affects

All elements, except menus.

See Also

IupZbox, IupVBox, IupHBox

POSITION (non inheritable)

The position of the element relative to the origin of the **Client** area of the native parent. If you add the CLIENTOFFSET attribute of the native parent, you can obtain the coordinates relative to the **Window** area of the native parent. See the Layout Guide.

It will be changed during the layout computation, except when FLOATING=YES or when used inside a concrete layout container.

Value

"x,y", where *x* and *y* are integer values corresponding to the horizontal and vertical position, respectively, in pixels.

Affects

All, except menus.

See Also

SIZE, RASTERSIZE, FLOATING, CLIENTOFFSET

SCREENPOSITION/X/Y (read-only) (non inheritable) (since 3.4)

Returns the absolute horizontal and/or vertical position of the top left corner of the client area relative to the origin of the main screen in pixels. It is similar to POSITION but relative to the origin of the main screen, instead of the origin of the client area. The origin of the main screen is at the upper left corner, in Windows it is affected by the position of the Start Menu when it is at the top or left side of the screen.

IMPORTANT: For the dialog, it is the position of the top left corner of the window, **NOT the client area**. It is the same position used in IupShowXY and IupPopup. In GTK, if the dialog is hidden the values can be outdated.

Value

"x,y", where *x* and *y* are integer values corresponding to the horizontal and vertical position, respectively, in pixels. When X or Y are used a single value is returned.

Affects

All controls that have visual representation.

See Also

POSITION

NAME (non inheritable) (since 3.0)

Name of the control inside the dialog. Not releated to IupSetHandle.

Value

Text.

Notes

The NAME value will be used by IupGetDialogChild to find a child inside a dialog.

Affects

All controls.

See Also

IupGetDialogChild

TIP (non inheritable)

Text to be shown when the mouse lies over the element.

Value

Text.

Additional Tip Attributes (since 3.0)

These attributes affect the TIP display.

TIPBALLOON [Windows Only]: The tip window will have the appearance of a cartoon "balloon" with rounded corners and a stem pointing to the item. Default: NO.

TIPBALLOONTITLE [Windows Only]: When using the balloon format, the tip can also has a title in a separate area.

TIPBALLOONTITLEICON [Windows Only]: When using the balloon format, the tip can also has a pre-defined icon in the title area. Values can be:

"0" - No icon (default)
"1" - Info icon
"2" - Warning icon
"3" - Error Icon

TIPBGCOLOR [Windows and Motif Only]: The tip background color. Default: "255 255 225" (Light Yellow)

TIPDELAY [Windows and Motif Only]: Time the tip will remain visible. Default: "5000". In Windows the maximum value is 32767 milliseconds.

TIPFGCOLOR [Windows and Motif Only]: The tip text color. Default: "0 0 0" (Black)

TIPFONT [Windows and Motif Only]: The font for the tip text. If not defined the font used for the text is the same as the FONT attribute for the element. If the value is SYSTEM then, no font is selected and the default system font for the tip will be used.

TIPICON [GTK only]: name of an image to be displayed in the TIP. See IupImage. (GTK 2.12)

TIPMARKUP [GTK only]: allows the tip string to contains Pango markup commands. Can be "YES" or "NO". Default: "NO". Must be set before setting the TIP attribute. (GTK 2.12)

TIPRECT (non inheritable): Specifies a rectangle inside the element where the tip will be activated. Format: "%d %d %d %d"="x1 y1 x2 y2". Default: all the element area. (GTK 2.12)

TIPVISIBLE: Shows or hides the tip under the mouse cursor. Use values "YES" or "NO". In GTK will only trigger the tip state, the given value will be ignored. Returns the current visible state. (GTK 2.12) (get since 3.5)

Additional Tip Callbacks (since 3.5)

TIPS_CB: Action before a tip is displayed.

```
int funcion(Ihandle* ih, int x, int y); [in C]
elem:action(x, y: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
x, y: cursor position relative to the top-left corner of the element

Affects

All controls that have visual representation, except menus.

TITLE (non inheritable)

Element's title. It is often used to modify some static text of the element (which cannot be changed by the user).

Value

Text.

Default: ""

Notes

The '\n' character usually is accepted for line change (except for menus).

The "&" character can be used to define a MNEMONIC, use "&&" to show the "&" character instead on defining a mnemonic.

If a mnemonic is defined then the character relative to it is underlined and a key is associated so that when pressed together with the Alt key activates the control.

In GTk, if you define a mnemonic using "&" and the string has an underscore, then make sure that the mnemonic comes before the underscore.

In GTK, if the MARKUP attribute is defined then the title string can contains pango markup commands. Works only if a mnemonic is NOT defined in the title. Not valid for menus.

Affects

All elements with an associated text.

See Also

FONT

VALUE (non inheritable)

Affects several elements differently - that is, its behavior is element dependent. It is often used to change the control's main value, such as the text of a IupText.

For the IupRadio and IupZbox, elements, which are categorized as composition elements, this attribute represents the element "selected" among the others in the designed composition. To change this attribute in such cases, different mechanisms are necessary according to the programming environment used. When the elements taking part in a composition were created in C, this attribute's contents is a name that must be defined by the IupSetHandle function. When the elements were created in Lua, this attribute's contents is the name of a variable - more precisely, the one receiving the return from the function that created the element you wish to select. In LED it is not possible to dynamically change the value of any attribute, so the elements created in this environment must be identified and manipulated in C by means of their identifying name.

WID (read-only) (non inheritable)

Element identifier in the native interface system.

Value

In Motif, returns the **Widget** handle.

In Windows, returns the **HWND** handle.

In GTK, return the **GtkWidget*** handle.

Notes

Verification-only attribute, available after the control is mapped.

For elements that do not have a native representation, NULL is returned.

Affects

All.

ZORDER (write-only) (non inheritable)

Change the ZORDER of a dialog or control. It is commonly used for dialogs, but it can be used to control the z-order of controls in a dialog.

Value

Can be "TOP" or "BOTTOM".

Affects

All controls that have visual representation.

DRAG & DROP (since 3.6)

When enabled allow the use of callbacks for controlling the drag and drop handling.

The user starts a drag and drop transfer by pressing the mouse button over the data (Windows and GTK: left button; Motif: middle button) which is referred to as the drag source. The data can be dropped in any location that has been registered as a drop target. The drop occurs when the user releases the mouse button. This can be done inside a control, from one control to another in the same dialog, in different dialogs of the same application, or between different applications (the other application does NOT need to be implemented with IUP).

In IUP, a drag and drop transfer can result in the data being moved or copied. A **copy** operation is enabled with the CTRL key pressed. A **move** operation is enabled with the SHIFT key pressed. A move operation will be possible only if the attribute DRAGSOURCEMOVE is Yes. When no key is pressed the default operation is **copy** when DRAGSOURCEMOVE=No and **move** when DRAGSOURCEMOVE=Yes. The user can cancel a drag at any time by pressing the ESCAPE key.

Steps to use the Drag & Drop support in an IUP application:

- **AT SOURCE:**
  - Enable the element as source using the attribute DRAGSOURCE=YES;
  - Define the data types supported by the element for the drag operation using the DRAGTYPES attribute;
  - Register the required callbacks DRAGBEGIN_CB, DRAGDATASIZE_CB and DRAGDATA_CB for drag handling. DRAGEND_CB is the only optional drag callback, all other callbacks and attributes must be set.
- **AT TARGET:**
  - Enable the element as target using the attribute DROPTARGET=YES;
  - Define the data types supported by the element for the drop using the DROPTYPES attribute;
  - Register the required callback DROPDATA_CB for handling the data received. This callback and all the drop target attributes must be set too. DROPMOTION_CB is the only optional drop callback.

Affects

IupLabel, IupText, IupList, IupTree, IupCanvas and IupDialog.

Attributes at Drag Source

DRAGCURSOR (non inheritable): name of an image to be used as cursor during drag. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. (since 3.11)

DRAGSOURCE (non inheritable): Set up a control as a source for drag operations. Default: NO.

DRAGTYPES (non inheritable): A list of data types that are supported by the source. Accepts a string with one or more names separated by commas. See Notes bellow for a list of known names. Must be set.

DRAGSOURCEMOVE (non inheritable): Enables the move action. Default: NO (only copy is enabled).

Attributes at Drop Target

DROPTARGET (non inheritable): Set up a control as a destination for drop operations. Default: NO.

DROPTYPES (non inheritable): A list of data types that are supported by the target. Accepts a string with one or more names separated by commas. See Notes bellow for a list of known names. Must be set.

Callbacks at Drag Source (Must be set when DRAGSOURCE=Yes)

DRAGBEGIN_CB: notifies source that drag started. It is called when the mouse starts a drag operation.

```
int function(Ihandle* ih, int x, int y) [in C]
elem:dragbegin_cb(x, y: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
x, y: cursor position relative to the top-left corner of the element.

Returns: If IUP_IGNORE is returned the drag is aborted.

DRAGDATASIZE_CB: request for size of drag data from source. It is called when the data is dropped, before the **DRAGDATA_CB** callback.

```
int function(Ihandle* ih, char* type) [in C]
elem:dragdatasize_cb(type: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
type: type of the data. It is one of the registered types in **DRAGTYPES**.

Returns: the size in bytes for the data. It will be used to allocate the buffer size for the data in transfer.

DRAGDATA_CB: request for drag data from source. It is called when the data is dropped.

```
int function(Ihandle* ih, char* type, void* data, int size) [in C]
elem:dragdata_cb(type: string, data: userdata, size: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
type: type of the data. It is one of the registered types in **DRAGTYPES**.
data: buffer to be filled by the application. In Lua is a light userdata.
size: buffer size in bytes. The same value returned by **DRAGDATASIZE_CB.**

DRAGEND_CB: notifies source that drag is done. The only drag callback that is **optional**. It is called after the data has been dropped.

```
int function(Ihandle* ih, int action) [in C]
elem:dragend_cb(action: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
action: action performed by the operation (1 = move, 0 = copy, -1 = drag failed or aborted)

If action is 1 it is responsibility of the application to remove the data from source.

Callbacks at Drop Target (Must be set when DROPTARGET=Yes)

DROPDATA_CB: source has sent target the requested data. It is called when the data is dropped. If both drag and drop would be in the same application it would be called after the **DRAGDATA_CB** callback.

```
int function(Ihandle* ih, char* type, void* data, int size, int x, int y) [in C]
elem:dropdata_cb(type: string, data: userdata, size, x, y: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
type: type of the data. It is one of the registered types in **DROPTYPES**.
data: content data received in the drop operation. In Lua is a light userdata.
size: data size in bytes.
x, y: cursor position relative to the top-left corner of the element.

DROPMOTION_CB: notifies destination about drag pointer motion. The only drop callback that is **optional**. It is called when the mouse moves over any valid drop site.

```
int function(Ihandle *ih, int x, int y, char *status); [in C]
elem:dropmotion_cb(x, y: number, status: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
x, y: position in the canvas where the event has occurred, in pixels.
status: status of mouse buttons and certain keyboard keys at the moment the event was generated. The same macros used for BUTTON_CB can be used for this status.

Notes

Drag and Drop support can be set independently. A control can have drop without drag support and vice-versa.

Here are some common Drag&Drop types defined by existing applications:

- "TEXT" used for regular text without formatting. Automatically translated to CF_TEXT in Windows.
- content MIME types, like "text/uri-list", "text/html", "image/png", "image/jpeg", "image/bmp" and "image/gif".
- "UTF8_STRING" in GTK and "UNICODETEXT" in Windows.
- "COMPOUND_TEXT" in GTK and "Rich Text Format" in Windows.
- "BITMAP" and "DIB" in Windows. Automatically translated to CF_BITMAP and CF_DIB.

Examples

list2.c

Global Attributes

Global attributes are not associated with any particular element. They represent and affect the global behavior of the toolkit.

To access global attributes use the IupGetGlobal and IupSetGlobal functions. In Lua global attributes can only be accessed trough those functions. In C, the functions IupGetAttribute and IupSetAttribute can also be used if you set the element handle to NULL.

General

LANGUAGE

The language used by some pre-defined dialogs.

Can have the values ENGLISH and PORTUGUESE. Default: ENGLISH. Can also be set by IupSetLanguage.

VERSION (read-only)

Returns the name of IUP's version.

The value follows the "major.minor.micro" format, major referring to broader changes, minor referring to smaller changes, and micro referring to corrections only. Ex.: "1.7.2".

COPYRIGHT (read-only)

Returns the IUP's copyright.

Ex: "Copyright (C) 1994-2014 Tecgraf/PUC-Rio".

DRIVER (read-only)

Informs the current driver being used.

Two drivers are available now, one for each platform: "GTK", "Motif" and "Win32".

System Control

LOCKLOOP

When the last visible dialog is closed the **IupExitLoop** function is called. To avoid that set LOCKLOOP=YES before hiding the last dialog. Possible values: "YES" or "NO". Default: "NO".

LASTERROR [Windows Only] (read-only) (since 3.6)

If an error is found, returns a string with the system error description.

UTF8MODE [Windows and GTK Only]

By default IUP uses strings in the current locale (See FONT attribute). To use UTF-8 strings set this attribute to Yes. Default: NO.

UTF8MODE_FILE [Windows Only]

By default IUP uses file names in the current locale, even when UTF8MODE=Yes. To use UTF-8 file names set this attribute to Yes. Default: NO.

DEFAULTPRECISION (since 3.11.2)

The default number of decimal places used in floating point output by some controls (**IupMatrixEx** and **IupGetParam**). Local attributes may overwrite the default. Default: 2.

DEFAULTDECIMALSYMBOL (since 3.13)

Symbol used for decimal separator in numeric values used in floating point output by some controls (**IupMatrixEx, IupGetParam** and **IupPlot**). Can be "." or "," only. Default uses the one defined by the system locale.

SHOWMENUIMAGES [GTK Only] (since 3.5)

Force the display of images in menus. Default: Yes

GLOBALMENU [GTK Only] (since 3.6)

Flag indicating that GTK is using a global menu instead of a per window menu. See more information at the GTK driver documentation.

GLOBALLAYOUTDLGKEY (since 3.17)

Flag to enable the global keys Alt+Ctrl+Shft+L to display the **IupLayoutDialog**.

GLOBALLAYOUTRESIZEKEY(since 3.17)

Flag to enable the global keys Ctrl+'+' and Ctrl+'-' that change the FONTSIZE and refresh the layout of the dialog. If element sizes are NOT set using RASTERSIZE their sizes will be automatically increased and decreased. Images are not changed.

IMAGEAUTOSCALE (since 3.16)

Automatically scale all images, except stock images, by a given real factor. If "DPI" value is used then the factor will be automatically calculated from the ratio between screen resolution and IMAGESDPI. Only images with BPP > 8 are scaled.

IMAGESDPI (since 3.16)

Defines the resolution of the images of the application. Common values are 96, 144, 192, and 288 DPI. Default: 96. Used when IMAGEAUTOSCALE=DPI.

IMAGESTOCKSIZE (since 3.16)

Force a size for stock images by controlling the image height. If that image size is not available the stock image is resized to match the given size. By default the size will be automatically calculated from the screen resolution: if res <= 96 DPI then size = 16, if 144 DPI size = 24, if 192 DPI size = 32, else size = 48. Only images with BPP > 8 are scaled.

IUPLUA_THREADED (since 3.6)

If defined allow IUP to be used inside coroutines in Lua.

SINGLEINSTANCE [Windows Only] (since 3.2)

Restricts the number of instances of the application by using a name to identify it. The value must also be a partial match to the title of a dialog that will receive the COPYDATA_CB callback with the command line of the second instance. When consulted returns NULL if inside the second instance. So usually in the application initialization after **IupOpen**, set SINGLEINSTANCE and then consult its value, if NULL abort the second instance by calling **IupClose** and returning from *main*.

System Mouse and Keyboard

CURSORPOS

Controls and returns the cursor position in absolute coordinates relative to the origin of the main screen. The origin of the main screen is at the upper left corner, in Windows it is affected by the position of the Start Menu when it is at the top or left side of the screen. Accept values in the format "X**x**Y" (in C "%dx%d), example "200x200". In GTK and Motif also generates mouse motion messages. (since GTK 2.8)

MOUSEBUTTON (write-only) (since 3.3)

Simulates a mouse button press, release or motion at the given coordinates. The position is in absolute coordinates relative to the upper left corner of the screen. Accept values in the format "X**x**Y button state" (in C "%dx%d %c %d"), example "200x200 1 1". **button** can be one of the IUP_BUTTON1,... definitions. **state** can be 2=double click, 1=pressed, 0=released, or -1=motion. The cursor position is always updated. In Windows button can be 'W' and state=delta, so a wheel button scroll is simulated.

IMPORTANT: not fully working. In Windows and GTK, menu items are not activated. Although submenus open, menu items even in the menu bar are not activated. In Windows, inside the **IupFileDlg** dialog, clicks in the folder navigation list are not correctly interpreted. In Motif click and drag operations are not performed.

SHIFTKEY (read-only) (since 3.0)

Returns the state of the Shift keys (left and right). Possible values: "ON" or "OFF".

CONTROLKEY (read-only) (since 3.0)

Returns the state of the Control keys (left and right). Possible values: "ON" or "OFF".

MODKEYSTATE (read-only) (since 3.0)

Returns the state of the keyboard modifier keys: Shift, Ctrl, Alt and sYs(Win/Apple). In the format of 4 characters: "SCAY". When not pressed the respective letter is replaced by a space " ".

KEYPRESS (write-only) (since 3.0)

Sends a key press message to the element with the focus. The value is a key code. See the Keyboard Codes table for a list of the possible values.

KEYRELEASE (write-only) (since 3.0)

Sends a key release message to the element with the focus. The value is a key code. See the Keyboard Codes table for a list of the possible values.

KEY (write-only) (since 3.0)

Sends a key press and a key release messages to the element with the focus. The value is a key code. See the Keyboard Codes table for a list of the possible values.

AUTOREPEAT [Motif Only]

Turns on/off ("YES" or "NO") the auto-repeat of keyboard keys in the whole system. May be used as an optimization in high performance applications.

INPUTCALLBACKS (since 3.4)

Turns on/off ("YES" or "NO") the global callbacks used to intercept global mouse and keyboard events. The callbacks must be set using the IupSetFunction function using the following names: **GLOBAL**KEYPRESS_CB, **GLOBAL**MOTION_CB, **GLOBAL**BUTTON_CB and **GLOBAL**WHEEL_CB (Windows Only). Their parameters are the same as the standard callbacks, but without the **Ihandle*** parameter.

In Lua use the iup.**SetGlobalCallback**(name, func) function. (since 3.7)

System Information

SYSTEM (read-only)

Informs the current operating system. On UNIX, it is equivalent to the command "uname -s" (sysname). On Windows, it identifies if you are on Windows 2000, Windows XP or Windows Vista. Some known names:

- "MacOS"
- "FreeBSD"
- "Linux"
- "SunOS"
- "Solaris"
- "IRIX"
- "AIX"
- "HP-UX"
- "Win2K"
- "WinXP"
- "Vista"
- "Win7"
- "Win8"

Notice that "Windows 8.1" will normally be detected as "Windows 8", unless a special Manifest is used. See MSDN for more information.

SYSTEMVERSION (read-only)

Informs the current operating system version number.

On UNIX, it is equivalent to the command "uname -r" (release). On Windows, it identifies the system version number and service pack version. On MacOSX is system version.

SYSTEMLANGUAGE (read-only)

Returns a text with a description of the system language.

SYSTEMLOCALE (read-only) (since 3.4)

Returns a text with a description of the system locale.

SCROLLBARSIZE (read-only) (since 3.9)

Returns the width of the vertical scrollbar (the same as the height of the horizontal scrollbar).

COMCTL32VER6 (read-only) [Windows Only] (since 3.11.1)

Returns Yes or No if the Windows common controls are using Visual Styles or not.

GTKVERSION (read-only) [GTK Only]

Returns the run time version of the GTK toolkit. This is the version being used at the time of the IupOpen function was called by the application.

GTKDEVVERSION (read-only) [GTK Only]

Returns the development version of the GTK toolkit. This is the version at the time the IUP library was compiled.

MOTIFVERSION (read-only) [Motif Only]

Returns the version of the run time Motif.

MOTIFNUMBER (read-only) [Motif Only]

Returns the number of the Motif Version if full form, e.x: 2.2.3 = "2203".

COMPUTERNAME (read-only)

Returns the hostname.

USERNAME (read-only)

Returns the user logged in.

EXEFILENAME (read-only)

Returns the filename of the executable with full path. Depending on how the program is executed the argv[0] not always has the full executable path.

GL_VERSION (read-only) (since 3.16)

Returns the OpenGL version. Available only after the first call to IupGLMakeCurrent.

GL_VENDOR (read-only) (since 3.16)

Returns the OpenGL vendor information. Available only after the first call to IupGLMakeCurrent.

GL_RENDERER (read-only) (since 3.16)

Returns the OpenGL renderer information. Available only after the first call to IupGLMakeCurrent.

XSERVERVENDOR (read-only) [GTK and Motif Only] (since 3.0)

X-Windows Server Vendor string.

XVENDORRELEASE (read-only) [GTK and Motif Only] (since 3.0)

X-Windows Server Vendor release number.

Screen Information

FULLSIZE (read-only)

Returns the full screen size in pixels.

String in the "*width*x*height*" format.

SCREENSIZE (read-only)

Returns the screen size in pixels available for dialogs, i.e. not including menu bars, task bars, etc. In Motif has the same value as the FULLSIZE attribute. The main screen size does not include additional monitors.

String in the "*width*x*height*" format.

SCREENDEPTH (read-only)

Returns the screen depth in bits per pixel.

SCREENDPI (read-only)

Returns a real value with the screen resolution in pixels per inch (same as dots per inch - DPI).

TRUECOLORCANVAS (read-only)

Indicates if the display allows creating TrueColor (> 8bpp) **IupCanvas** controls, even if PseudoColor is the default, i.e. even if SCREENDEPTH<=8 . Returns "YES" or "NO". Usefull in Motif.

DWM_COMPOSITION (read-only) [Win32 Only] (since 3.10)

Returns the Desktop Window Manager Composition flag. Returns "YES" or "NO". Works only in Windows Vista and newer. Returns NULL if not supported.

VIRTUALSCREEN (read-only) [Win32 and GTK Only] (since 3.0)

Returns the virtual screen position and size in pixels. It is the virtual space defined by all monitors in the system.

String in the "*x y width height*" format.

MONITORSCOUNT (read-only) [Win32 and GTK Only] (since 3.17)

Returns the number of monitors.

MONITORSINFO (read-only) [Win32 and GTK Only] (since 3.0)

Returns the position and size in pixels of all monitors. Each monitor information is terminated by a "\n" character.

String in the "*x y width height*\n*x y width height*\n..." format.

System Data

HINSTANCE (read-only) [Win32 Only]

Returns a handle (HINSTANCE) that identifies the application in the native system.

DLL_HINSTANCE [Win32 Only] (since 3.0)

Changes and returns a handle (HINSTANCE) that identifies the DLL where resources are stored.

APPSHELL (read-only) [Motif Only] (since 3.0)

Returns the shell Widget created by XtOpenApplication.

XDISPLAY (read-only) [GTK and Motif Only] (since 3.0)

Returns the X-Windows Display.

XSCREEN (read-only) [GTK and Motif Only] (since 3.0)

Returns the X-Windows Screen.

Default Attributes

DLGBGCOLOR

The default background color for all elements that have the background similar of the dialog.

DLGFGCOLOR (since 3.0)

The default foreground color for all elements that have text over the background of the dialog or similar. Usually is "0 0 0".

MENUBGCOLOR [Windows Only] (since 3.0)

The default menu background color. Usually is "255 255 255".

MENUFGCOLOR [Windows Only] (since 3.0)

The system default menu foreground color. Usually is "0 0 0". Unused in IUP.

TXTBGCOLOR (since 3.0)

The default background color for editable text, also used by lists and tree. Usually is "255 255 255".

TXTFGCOLOR (since 3.0)

The default foreground color for editable text, also used by lists and tree. Usually is "0 0 0".

TXTHLCOLOR (since 3.16)

The default highlight color for editable text, also used by lists and tree. The highlight color is used when the text is selected. Usually is "0 0 0" in Motif, and "51 153 255" in Windows. Can be changed only in **IupTree**, and only in Windows and Motif. But it can be used for drawing selected areas in custom controls.

LINKFGCOLOR (since 3.8)

The default foreground color for linked text. In GTK and Motif is "0 0 238".

DEFAULTFONT

The default font used by all elements, except for menus.

DEFAULTFONTFACE (since 3.13)

Auxiliary attribute to retrieve and set the default font face used by all elements. It retrieves the typeface from DEFAULTFONT. When changed will actually change the DEFAULTFONT.

DEFAULTFONTSIZE (since 3.0)

Auxiliary attribute to retrieve and set the default font size used by all elements. It retrieves the size from DEFAULTFONT. When changed will actually change the DEFAULTFONT.

DEFAULTFONTSTYLE (since 3.11)

Auxiliary attribute to retrieve and set the default font style used by all elements. It retrieves the style from DEFAULTFONT. When changed will actually change the DEFAULTFONT.

DEFAULTBUTTONPADDING (since 3.16)

Default button padding used in pre-defined dialogs. Default: 12x4".

Events and Callbacks

IUP is a graphics interface library, so most of the time it waits for an event to occur, such as a button click or a mouse leaving a window. The application can inform IUP which callback to be called, informing that an event has taken place. Hence events are handled through callbacks, which are just functions that the application register in IUP.

The events are processed only when IUP has the control of the application. After the application creates and shows a dialog it must return the control to IUP so it can process incoming events. This is done in the IUP main event loop. And it is usually done once at the application "main" function. One exception is the display of modal dialogs. These dialogs will have their own event loop and the previous shown dialogs will stop receiving events until the modal dialog returns.

Events and Callbacks Guide

Using

Callbacks are used by the application to receive notifications from the system that the user or the system itself has interacted with the user interface of the application. On the other hand attributes are used by the application to communicate with the user interface system.

Even though callbacks have different purposes from attributes, they are also associated to an element by means of an name.

The OLD method to associate a function to a callback, the application must employ the **IupSetAttribute** function, linking the action to a name (passed as a string). From this point on, this name will refer to a callback. By means of function **IupSetFunction**, the user connects this name to the callback. For example:

```
int myButton_action(Ihandle* self);
...
IupSetAttribute(myButton, "ACTION", "my_button_action");
IupSetFunction("my_button_action", (Icallback)myButton_action);
```

In LED, callback are only assigned by their names. It will be still necessary to associate the name with the corresponding function in C using **IupSetFunction**. For example:

```
# In LED, is equivalent to the IupSetAttribute command in the previous example.
bt = button("Title", my_button_action)
```

In the NEW method, the application does not needs a global name, it directly sets the callback using the attribute name using **IupSetCallback**. For example:

```
int myButton_action(Ihandle* self);
...
IupSetCallback(myButton, "ACTION", (Icallback)myButton_action);
```

The new method is more efficient and more secure, because there is no risk of a name conflict. If the application uses LED, just ignore the name in the LED, and replace **IupSetFunction** by **IupSetCallback**.

Although enabled in old versions, callbacks do NOT have **inheritance** like attributes.

All callbacks receive at least the element which activated the action as a parameter (self).

The callbacks implemented in C by the application must return one of the following values:

- IUP_DEFAULT: Proceeds normally with user interaction. In case other return values do not apply, the callback should return this value.
- IUP_CLOSE: Call **IupExitLoop** after return. Depending on the state of the application it will close all windows and exit the application. Applies only to some actions.
- IUP_IGNORE: Makes the native system ignore that callback action. Applies only to some actions.
- IUP_CONTINUE: Makes the element to ignore the callback and pass the treatment of the execution to the parent element. Applies only to some actions.

Only some callbacks support the last 3 return values. Check each callback documentation. When nothing is documented then only IUP_DEFAULT is supported.

An important detail when using callbacks is that they are only called when the user actually executes an action over an element. A callback is not called when the programmer sets a value via **IupSetAttribute**. For instance: when the programmer changes a selected item on a list, no callback is called.

The order of callback calling is system dependent. For instance, the RESIZE_CB and the SHOW_CB are called in different order in Win32 and in X-Windows when the dialog is shown for the first time.

To help the definition of callbacks in C, the header "iupcbs.h" can be used, there are typedefs for all the callbacks.

Main Loop

IUP is an event-oriented interface system, so it will keep a loop "waiting" for the user to interact with the application. For this loop to occur, the application must call the **IupMainLoop** function, which is generally used right before **IupClose**.

When the application is closed by returning IUP_CLOSE in a callback, calling **IupExitLoop** or by hiding the last visible dialog, the function **IupMainLoop** will return.

The **IupLoopStep** and the **IupFlush** functions force the processing of incoming events while inside an application callback.

IupLua

Callbacks in Lua have the same names and receive the same parameters as callbacks in C, in the same order. In Lua the callbacks they can either return a value or not, the IupLua binding will automatically return IUP_DEFAULT if no value is returned. In Lua callbacks can be Lua functions or strings with Lua code.

The callbacks can also be implemented as methods, using the language's resources for object orientation. Thus, the element is implicitly passed as the **self** parameter.

The following example shows the definition of an action for a button.

```
function myButton:action ()
  local aux = self.fgcolor
  self.fgcolor = self.bgcolor
  self.bgcolor = aux
end
```

Or you can do

```
function myButton_action(self)
  ...
end
myButton.action = myButton_action
```

Or also

```
myButton.action = function (self)
  ...
end
```

Or, as a string

```
myButton.action = "local aux = self.fgcolor;
                   self.fgcolor = self.bgcolor;
                   self.bgcolor = aux"
```

Altough some callbacks exists only in specific controls, all the callbacks can be set for all the controls. This is usefull to set a callback for a box, so it will be inherited by all the elements inside that box which implements that callback.

IupMainLoop

Executes the user interaction until a callback returns IUP_CLOSE, **IupExitLoop** is called, or hiding the last visible dialog.

Parameters/Return

```
int IupMainLoop(void); [in C]
iup.MainLoop() -> ret: number [in Lua]
```

Returns: IUP_NOERROR always.

Notes

When this function is called, it will interrupt the program execution until a callback returns IUP_CLOSE, **IupExitLoop** is called, or there are no visible dialogs.

If you cascade many calls to **IupMainLoop** there must be a "return IUP_CLOSE" or **IupExitLoop** call for each cascade level, hiddinh all dialogs will close only one level. Call IupMainLoopLevel to obtain the current level.

If **IupMainLoop** is called without any visible dialogs and no active timers, the application will hang and will not be possible to close the main loop. The process will have to be interrupted by the system.

When the last visible dialog is hidden the **IupExitLoop** function is automatically called, causing the **IupMainLoop** to return. To avoid that set LOCKLOOP=YES before hiding the last dialog.

See Also

IupOpen, IupClose, IupLoopStep, IupExitLoop, Guide/System Control, IDLE_ACTION, LOCKLOOP.

IupMainLoopLevel (since 3.0)

Returns the current cascade level of **IupMainLoop**. When no calls were done, return value is 0.

Parameters/Return

```
int IupMainLoopLevel(void); [in C]
iup.MainLoopLevel() -> ret: number [in Lua]
```

Returns: the cascade level.

Notes

You can use this function to check if **IupMainLoop** was already called and avoid calling it again.

A call to **IupPopup** will increase one level.

See Also

IupOpen, IupClose, IupLoopStep, Guide/System Control, IDLE_ACTION, LOCKLOOP.

IupLoopStep

Runs one iteration of the message loop.

Parameters/Return

```
int IupLoopStep(void); [in C]
int IupLoopStepWait(void); [in C]

iup.LoopStep() -> ret: number [in Lua]
iup.LoopStepWait() -> ret: number [in Lua]
```

Returns: IUP_CLOSE or IUP_DEFAULT.

Notes

This function is useful for allowing a second message loop to be managed by the application itself. This means that messages can be intercepted and callbacks can be processed inside an application loop.

IupLoopStep returns immediately after processing any messages or if there are no messages to process. **IupLoopStepWait** put the system in idle until a message is processed (since 3.0).

If IUP_CLOSE is returned the **IupMainLoop** will not end because the return code was already processed. If you want to end **IupMainLoop** when IUP_CLOSE is returned by **IupLoopStep** then call **IupExitLoop** after **IupLoopStep** returns.

An example of how to use this function is a counter that can be stopped by the user. For such, the user has to interact with the system, which is possible by calling the function periodically.

This way, this function replaces old mechanisms implemented using the Idle callback.

Note that this function does not replace **IupMainLoop**.

See Also

IupOpen, IupClose, IupMainLoop, IupExitLoop, IDLE_ACTION, Guide / System Control

IupExitLoop

Terminates the current message loop. It has the same effect of a callback returning IUP_CLOSE.

Parameters/Return

```
void IupExitLoop(void); [in C]
iup.ExitLoop() [in Lua]
```

IupFlush

Processes all pending messages in the message queue.

Parameters/Return

```
void IupFlush(void); [in C]
iup.Flush() [in Lua]
```

Notes

When you change an attribute of a certain element, the change may not take place immediately. For this update to occur faster than usual, call **IupFlush** after the attribute is changed.

Important: A call to this function may cause other callbacks to be processed before it returns.

In Motif, if the X server sent an event which is not yet in the event queue, after a call to **IupFlush** the queue might not be empty.

IupGetCallback

Returns the callback associated to an event.

Parameters/Return

```
Icallback IupGetCallback(Ihandle* ih, const char *name); [in C]
[There is no equivalent in Lua]
```

ih: identifier of the interface element.
name: attribute name of the callback.

Returns: the callback or NULL if there is none.

Notes

This function replaces the deprecated combination:

```
IupGetFunction(IupGetAttribute(ih, name))
```

If an event is associated using **IupSetFunction** and **IupSetAttribute**, the **IupGetCallback** also returns the correct callback. So old applications work normally.

See Also

IupSetCallback, IupGetFunction

IupSetCallback

Associates a callback to an event.

Parameters/Return

```
Icallback IupSetCallback(Ihandle* ih, const char *name, Icallback func); [in C]
```

```
[There is no equivalent in Lua]
```

ih: identifier of the interface element.
name: name of the callback.
func: address of a C function. If NULL removes the association.

Returns: the address of the previous function associated to the action.

Notes

This function replaces the deprecated combination:

```
IupSetFunction(global_name, func);
IupSetAttribute(ih, name, global_name);
```

So it eliminates the need for a global name.

Callbacks set using **IupSetCallback** can not be retrieved using **IupGetFunction**.

In Lua, callbacks are associated by simply setting a function as the value of the callback name, for example:

```
button = iup.button{...

button.action = function(...    OR
function button:action(...
```

See Also

IupGetCallback, IupSetFunction

IupSetCallbacks

Associates several callbacks to an event.

Parameters/Return

```
Ihandle* IupSetCallbacks(Ihandle* ih, const char *name, Icallback func, ...); [in C]
[There is no equivalent in Lua]
```

ih: identifier of the interface element.
name: name of the callback.
func: address of a C function. If NULL removes the association.

Returns: the same **ih** handle.

Notes

It is useful for setting many callbacks at once while in the creation of an hierarchy of elements, just like **IupSetAttributes**.

See Also

IupSetCallback, IupSetAttributes

IupGetActionName (Deprecated since 3.0)

Should return the name of the action being executed by the application. It works **only** if the application used IupSetFunction, which is now **deprecated**.

Parameters/Return

```
const char* IupGetActionName(void); [in C]
[There is no equivalent in Lua]
```

Returns: the name of the action.

See Also

DEFAULT_ACTION, IupSetFunction

IupGetFunction

Returns the function associated to an action only when they were set by IupSetFunction. It will not work if IupSetCallback were used.

Parameters/Return

```
Icallback IupGetFunction(const char *name); [in C]
[There is no equivalent in Lua]
```

name: name of the action.

Returns: the callback or NULL if not found.

See Also

IupSetFunction, IupGetCallback

IupSetFunction

Associates a function to an action as a global callback.

This function should not be used by new applications, use it only for global callbacks. For regular elements use IupSetCallback instead.

Notice that the application or libraries may set the same name for two different functions by mistake. **IupSetCallback** does not depends on global names.

Parameters/Return

```
Icallback IupSetFunction(const char *name, Icallback func); [in C]
[There is no equivalent in Lua]
```

name: name of an action.
func: address of a C function. If NULL removes the association.

Returns: the address of the previous function associated to the action.

See Also

IupGetFunction, IupSetCallback,

IupRecordInput

Records all mouse and keyboard input in a file for later reproduction.

Parameters/Return

```
int IupRecordInput(const char *filename, int mode); [in C]
iup.RecordInput(filename: string, mode: number) -> ret: number [in Lua]
```

filename: name of the file to be saved. NULL will stop recording.
mode: flag for controlling the file generation. Can be: IUP_RECBINARY or IUP_RECTEXT.

Returns: IUP_NOERROR if successful, IUP_ERROR if failed to open the file for writing.

Notes

Any existing file will be replaced.

Must stop recording before exiting the application.

It uses the global callbacks enabled by the INPUTCALLBACKS global attribute.

Mouse position is relative to the top left corner of the screen and it is independent from the controls and dialogs being manipulated.

The generated file can be used by **IupPlayInput** to reproduce the same events.

See Also

INPUTCALLBACKS, IupPlayInput

IupPlayInput

Reproduces all mouse and keyboard input from a given file.

Parameters/Return

```
int IupPlayInput(const char *filename); [in C]
iup.PlayInput(filename: string) -> ret: number [in Lua]
```

filename: name of the file to be played. NULL will stop playing. "" will pause and restart a file already being played.

Returns: IUP_NOERROR if successful, IUP_ERROR if failed to open the file for writing. If already playing

Notes

The file must had been saved using the **IupRecordInput** function. Record mode will be automatically detected.

This function will start the play and return the control to the application. If the file ends all internal memory used to play the file will be automatically released.

It uses the MOUSEBUTTON global attribute to reproduce the events. **IMPORTANT**: See the documentation of the MOUSEBUTTON attribute for further details and current limitations.

The file must had been generated in the same operating system. Screen size differences can exist, but if different themes are used then mouse precision will be affected.

See Also

MOUSEBUTTON, IupRecordInput

DEFAULT_ACTION (Deprecated since 3.0)

Predefined IUP action, generated every time an action has no associated function (except for the IDLE_ACTION).

Callback

```
int function(Ihandle *ih); [in C]
[There is no Lua equivalent]
```

ih: identifier of the element that activated the function.

Notes

Often a programmer defines an action with a name and, when associating it to a function, he/she mistypes the action name, or vice-versa. This kind of mistake is very common, and IUP is not able to automatically detect it. This predefined action, combined with function **IupGetActionName**, can help the programmer detect this problem. All you have to do is define a default action and verify which is the name of the action that activated it. For example:

```
IupSetFunction("myFunctionName", (Icallback)myFunction);
IupSetAttribute(myButton, "ACTION", "myFunctionNamr");   /* notice the typo error here */
```

In this case the incorrect name "myFunctionNamr" (typo error here) will not be found, so if the DEFAULT_ACTION is defined it will be called when "ACTION" in invoked for the button. In fact it will be called for all the actions that do not have an action associated.

Affects

All callbacks when **IupSetFunction** is used. If **IupSetCallback** is used DEFAULT_ACTION is ignored.

See Also

IupSetFunction, IupGetActionName.

IDLE_ACTION

Predefined IUP action, generated when there are no events or messages to be processed. Often used to perform background operations.

Callback

```
int function(void); [in C]
```

Returns: if IUP_CLOSE is returned the current loop will be closed and the callback will be removed. If IUP_IGNORE is returned the callback is removed and normal processing continues.

Notes

The Idle callback will be called whenever there are no messages left to be processed. But this occurs more frequent than expected, for example if you move the mouse over the application the idle callback will be called many times because the mouse move message is processed so fast that the Idle will be called before another mouse move message is schedule to processing.

So this callback changes the message loop to a more CPU consuming one. It is important that you set it to NULL when not using it. Or the application will be consuming CPU even if the callback is doing nothing.

It can only be set using **IupSetFunction(**name, func**)**.

Lua Binding

To modify this action use the function **iup.SetIdle(**func**)** in Lua. Using nil as a parameter to remove the association.

Or use the iup.**SetGlobalCallback**(name, func) function. (since 3.7)

Long Time Operations

If you create a loop or an operation that takes a long time to complete inside a callback of your application then the user interface message loop processing is interrupted until the callback returns, so the user can not click on any control of the application. But there are ways to handle that:

- call **IupLoopStep** or **IupFlush** inside the application callback when it is performing long time operations. This will allow the user to click on a cancel button for instance, because the user interface message loop will be processed.
- split the operation in several parts that are processed by the **Idle** function when no messages are left to be processed for the user interface message loop. This will make a heavy use of the CPU, even if the callback is doing nothing.
- split the operation in several parts but use a **Timer** to process each part.

If you just want to do something simple as a background redraw of an **IupCanvas**, then a better idea is to handle the "idle" state yourself. For example, register a timer for a small time like 500ms, and reset the timer in all the mouse and keyboard callbacks of the **IupCanvas**. If the timer is trigged then you are in idle state. If the **IupCanvas** loses its focus then stop the timer.

Examples

Browse for Example Files

See Also

IupSetFunction, IupTimer.

MAP_CB

Called right after an element is mapped and its attributes updated in IupMap.

When the element is a dialog, it is called after the layout is updated. For all other elements is called before the layout is updated, so the element current size will still be 0x0 during MAP_CB (since 3.14).

Callback

```
int function(Ihandle *ih); [in C]
ih:map_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Affects

All that have a native representation.

UNMAP_CB

Called right before an element is unmapped.

Callback

```
int function(Ihandle *ih); [in C]
ih:unmap_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Affects

All that have a native representation.

DESTROY_CB

Called right **before** an element is destroyed.

Callback

```
int function(Ihandle *ih); [in C]
ih:destroy_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Notes

If the dialog is visible then it is hidden before it is destroyed. The callback will be called right **after** it is hidden.

The callback will be called **before** all other destroy procedures.

For instance, if the element has children then it is called **before** the children are destroyed.

For language binding implementations use the callback name "LDESTROY_CB" to release memory allocated by the binding for the element. Also the callback will be called **before** the language callback.

Affects

All.

GETFOCUS_CB

Action generated when an element is given keyboard focus. This callback is called after the KILLFOCUS_CB of the element that loosed the focus. The IupGetFocus function during the callback returns the element that loosed the focus.

Callback

```
int function(Ihandle *ih); [in C]
ih:getfocus_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that received keyboard focus.

Affects

All elements with user interaction, except menus.

See Also

KILLFOCUS_CB, IupGetFocus, IupSetFocus

KILLFOCUS_CB

Action generated when an element loses keyboard focus. This callback is called before the GETFOCUS_CB of the element that gets the focus.

Callback

```
int function(Ihandle *ih); [in C]
ih:killfocus_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Affects

All elements with user interaction, except menus.

In Windows, there are restrictions when using this callback. From MSDN on WM_KILLFOCUS: ""While processing this message, do not make any function calls that display or activate a window. This causes the thread to yield control and can cause the application to stop responding to messages."

See Also

GETFOCUS_CB, IupGetFocus, IupSetFocus

ENTERWINDOW_CB

Action generated when the mouse enters the native element.

Callback

```
int function(Ihandle *ih); [in C]
ih:enterwindow_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Notes

When the cursor is moved from one element to another, the call order in all platforms will be first the LEAVEWINDOW_CB callback of the old control followed by the ENTERWINDOW_CB callback of the new control. (since 3.14)

Affects

All controls with user interaction.

See Also

LEAVEWINDOW_CB

LEAVEWINDOW_CB

Action generated when the mouse leaves the native element.

Callback

```
int function(Ihandle *ih); [in C]
ih:leavewindow_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Notes

When the cursor is moved from one element to another, the call order in all platforms will be first the LEAVEWINDOW_CB callback of the old control followed by the ENTERWINDOW_CB callback of the new control. (since 3.14)

Affects

All controls with user interaction.

See Also

ENTERWINDOW_CB

K_ANY

Action generated when a keyboard event occurs.

Callback

```
int function(Ihandle *ih, int c); [in C]
ih:k_any(c: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
c: identifier of typed key. Please refer to the Keyboard Codes table for a list of possible values.

Returns: If IUP_IGNORE is returned the key is ignored and not processed by the control and not propagated. If returns IUP_CONTINUE, the key will be processed and the event will be propagated to the parent of the element receiving it, this is the default behavior. If returns IUP_DEFAULT the key is processed but it is not propagated. IUP_CLOSE will be processed.

Notes

Keyboard callbacks depend on the keyboard usage of the control with the focus. So if you return IUP_IGNORE the control will usually not process the key. But be aware that sometimes the control process the key in another event so even returning IUP_IGNORE the key can get processed. Although it will not be propagated.

IMPORTANT: The callbacks "K_*" of the dialog or native containers depend on the IUP_CONTINUE return value to work while the control is in focus.

If the callback does not exists it is automatically propagated to the parent of the element.

K_* callbacks

All defined keys are also callbacks of any element, called when the respective key is activated. For example: "K_cC" is also a callback activated when the user press Ctrl+C, when the focus is at the element or at a children with focus. This is the way an application can create shortcut keys, also called hot keys. These callbacks are not available in IupLua.

Affects

All elements with keyboard interaction.

HELP_CB

Action generated when the user press F1 at a control. In Motif is also activated by the Help button in some workstations keyboard.

Callback

```
void function(Ihandle *ih); [in C]
ih:help_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Returns: IUP_CLOSE will be processed.

Affects

All elements with user interaction.

ACTION

Action generated when the element is activated. Affects each element differently.

Callback

```
int function(Ihandle *ih); [in C]
ih:action() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

In some elements, this callback may receive more parameters, apart from **ih**. Please refer to each element's documentation.

Affects

IupButton, IupItem, IupList, IupText, IupCanvas, IupMultiline, IupToggle

Dialogs

In IUP you can create your own dialogs or use one of the predefined dialogs. To create your own dialogs you will have to create all the controls of the dialog before the creation of the dialog. All the controls must be composed in a hierarchical structure so the root will be used as a parameter to the dialog creation.

When a control is created, its parent is not known. After the dialog is created all elements receive a parent. This mechanism is quite different from that of native systems, who first create the dialog and then the element are inserted, using the dialog as a parent. This feature creates some limitations for IUP, usually related to the insertion and removal of controls.

Since the controls are created in a different order from the native system, native controls can only be created after the dialog. This will happen automatically when the application call the **IupShow** function to show the dialog. But we often need the native controls to be created so we can use some other functionality of those before they are visible to the user. For that purpose, the **IupMap** function was created. It forces IUP to map the controls to their native system controls. The **IupShow** function internally uses **IupMap** before showing the dialog on the screen. **IupShow** can be called many times, but the map process will occur only once.

IupShow can be replaced by **IupPopup**. In this case the result will be a modal dialog and all the other previously shown dialogs will be unavailable to the user. Also the program will interrupt in the function call until the application return IUP_CLOSE or **IupExitLoop** is called.

All dialogs are automatically destroyed in **IupClose**.

IupDialog

Creates a dialog element. It manages user interaction with the interface elements. For any interface element to be shown, it must be encapsulated in a dialog.

Creation

```
Ihandle* IupDialog(Ihandle *child); [in C]
iup.dialog{child: ihandle} -> (elem: ihandle) [in Lua]
dialog(child) [in LED]
```

child: Identifier of an interface element. The dialog has only one child. It can be NULL (nil in Lua), not optional in LED.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

Common

BACKGROUND (non inheritable): Dialog background color or image. Can be a non inheritable alternative to BGCOLOR or can be the name of an image to be tiled on the background. See also the screenshots of the sample.c results with normal background, changing the dialog BACKGROUND, the dialog BGCOLOR and the children BGCOLOR. Not working in GTK 3. (since 3.0)

BORDER (non inheritable) (creation only): Shows a resize border around the dialog. Default: "YES". BORDER=NO is useful only when RESIZE=NO, MAXBOX=NO, MINBOX=NO, MENUBOX=NO and TITLE=NULL, if any of these are defined there will be always some border.

CHILDOFFSET: Allow to specify a position offset for the child. Available for native containers only. It will not affect the natural size, and allows to position controls outside the client area. Format "*dx*x*dy*", where *dx* and *dy* are integer values corresponding to the horizontal and vertical offsets, respectively, in pixels. Default: 0x0. (since 3.14)

CURSOR (non inheritable): Defines a cursor for the dialog.

DROPFILESTARGET [Windows and GTK Only] (non inheritable): Enable or disable the drop of files. Default: NO, but if DROPFILES_CB is defined when the element is mapped then it will be automatically enabled.

EXPAND (non inheritable): The default value is "YES".

NACTIVE (non inheritable): same as ACTIVE but does not affects the controls inside the dialog. (since 3.13)

SIZE (non inheritable): Dialog's size. Additionally the following values can also be defined for width and/or height:

- "FULL": Defines the dialog's width (or height) equal to the screen's width (or height)
- "HALF": Defines the dialog's width (or height) equal to half the screen's width (or height)
- "THIRD": Defines the dialog's width (or height) equal to 1/3 the screen's width (or height)
- "QUARTER": Defines the dialog's width (or height) equal to 1/4 of the screen's width (or height)
- "EIGHTH": Defines the dialog's width (or height) equal to 1/8 of the screen's width (or height)

The dialog **Natural** size is only considered when the **User** size is not defined or when it is bigger than the **Current** size. This behavior is different from a control that goes inside the dialog. Because of that, when SIZE or RASTERSIZE are set (changing the **User** size), the **Current** size is internally reset to 0x0, so the the **Natural** size can be considered when re-computing the **Current** size of the dialog.

Values set at SIZE or RASTERSIZE attributes of a dialog are always accepted, regardless of the minimum size required by its children. For a dialog to have the minimum necessary size to fit all elements contained in it, simply define SIZE or RASTERSIZE to NULL. Also if you set SIZE or RASTERSIZE to be used as the initial size of the dialog, its contents will be limited to this size as the minimum size, if you do not want that, then after showing the dialog reset this size to NULL so the dialog can be resized to smaller values. But notice that its contents will still be limited by the **Natural** size, to also remove that limitation set SHRINK=YES. To only change the **User** size in pixels, without resetting the **Current** size, set the USERSIZE attribute (since 3.12).

Notice that the dialog size includes its decoration (it is the **Window** size), the area available for controls are returned by the dialog CLIENTSIZE. For more information see Layout Guide.

TITLE (non inheritable): Dialog's title. Default: NULL. If you want to remove the title bar you must also set MENUBOX=NO, MAXBOX=NO and MINBOX=NO, before map. But in Motif and GTK it will hide it only if RESIZE=NO also.

VISIBLE: Simply call **IupShow** or **IupHide** for the dialog.

ACTIVE, BGCOLOR, FONT, EXPAND, SCREENPOSITION, WID, TIP, CLIENTSIZE, RASTERSIZE, ZORDER: also accepted. Note that ACTIVE, BGCOLOR and FONT will also affect all the controls inside the dialog.

Drag & Drop attributes and callbacks are supported.

Exclusive

DEFAULTENTER: Name of the button activated when the user press Enter when focus is in another control of the dialog. Use IupSetHandle or IupSetAttributeHandle to associate a button to a name.

DEFAULTESC: Name of the button activated when the user press Esc when focus is in another control of the dialog. Use IupSetHandle or IupSetAttributeHandle to associate a button to a name.

DIALOGFRAME: Set the common decorations for modal dialogs. This means RESIZE=NO, MINBOX=NO and MAXBOX=NO. In Windows, if the PARENTDIALOG is defined then the MENUBOX is also removed, but the Close button remains.

ICON: Dialog's icon. The Windows SDK recommends that cursors and icons should be implemented as resources rather than created at run time.

FULLSCREEN: Makes the dialog occupy the whole screen over any system bars in the main monitor. All dialog details, such as title bar, borders, maximize button, etc, are removed. Possible values: YES, NO. In Motif you may have to click in the dialog to set its focus.
In Motif if set to YES when the dialog is hidden, then it can not be changed after it is visible.

HWND [Windows Only] (non inheritable, read-only): Returns the Windows Window handle. Available in the Windows driver or in the GTK driver in Windows.

MAXBOX (creation only): Requires a maximize button from the window manager. If RESIZE=NO then MAXBOX will be set to NO. Default: YES. In Motif the decorations are controlled by the Window Manager and may not be possible to be changed from IUP. In Windows MAXBOX is hidden only if MINBOX is hidden as well, or else it will be just disabled.

MAXSIZE: Maximum size for the dialog in raster units (pixels). The windowing system will not be able to change the size beyond this limit. Default: 65535x65535. (since 3.0)

MENU: Name of a menu. Associates a menu to the dialog as a menu bar. The previous menu, if any, is unmapped. Use IupSetHandle or IupSetAttributeHandle to associate a menu to a name. See also IupMenu.

MENUBOX (creation only): Requires a system menu box from the window manager. If hidden will also remove the Close button. Default: YES. In Motif the decorations are controlled by the Window Manager and may not be possible to be changed from IUP. In Windows if hidden will hide also MAXBOX and MINBOX.

MINBOX (creation only): Requires a minimize button from the window manager. Default: YES. In Motif the decorations are controlled by the Window Manager and may not be possible to be changed from IUP. In Windows MINBOX is hidden only if MAXBOX is hidden as well, or else it will be just disabled.

MINSIZE: Minimum size for the dialog in raster units (pixels). The windowing system will not be able to change the size beyond this limit. Default: 1x1. Some systems define a very minimum size greater than this, for instance in Windows the horizontal minimum size includes the window decoration buttons. (since 3.0)

MODAL (read-only): Returns the popup state. It is "YES" if the dialog was shown using **IupPopup**. It is "NO" if **IupShow** was used or it is not visible. At the first time the dialog is shown, MODAL is not set yet when SHOW_CB is called. (since 3.0)

NATIVEPARENT (creation only): Native handle of a dialog to be used as parent. Used only if PARENTDIALOG is not defined.

PARENTDIALOG (creation only): Name of a dialog to be used as parent.

PLACEMENT: Changes how the dialog will be shown. Values: "FULL", "MAXIMIZED", "MINIMIZED" and "NORMAL". Default: NORMAL. After **IupShow**/**IupPopup** the attribute is set back to "NORMAL". FULL is similar to FULLSCREEN but only the dialog client area covers the screen area, menu and decorations will be there but out of the screen. In UNIX there is a chance that the placement won't work correctly, that depends on the Window Manager. In Windows, the SHOWNOACTIVE attribute can be set to Yes to avoid to make the window active (since 3.15). In Windows, the SHOWMINIMIZENEXT attribute can be set to Yes to activate the next top-level window in the Z order when minimizing (since 3.15).

RESIZE (creation only): Allows interactively changing the dialog's size. Default: YES. If RESIZE=NO then MAXBOX will be set to NO. In Motif the decorations are controlled by the Window Manager and may not be possible to be changed from IUP.

SAVEUNDER [Windows and Motif Only] (creation only): When this attribute is true (YES), the dialog stores the original image of the desktop region it occupies (if the system has enough memory to store the image). In this case, when the dialog is closed or moved, a redrawing event is not generated for the windows that were shadowed by it. Its default value is YES. To save memory disable it for your main dialog. Not available in GTK.

SHRINK: Allows changing the elements' distribution when the dialog is smaller than the minimum size. Default: NO.

STARTFOCUS: Name of the element that must receive the focus right after the dialog is shown using **IupShow** or **IupPopup**. If not defined then the first control than can receive the focus is selected (same effect of calling IupNextField for the dialog). Updated after SHOW_CB is called and only if the focus was not changed during the callback.

XWINDOW [UNIX Only] (non inheritable, read-only): Returns the X-Windows Window (Drawable). Available in the Motif driver or in the GTK driver in UNIX.

Exclusive [Windows and GTK Only]

ACTIVEWINDOW [Windows and GTK Only] (read-only): informs if the dialog is the active window (the window with focus). Can be Yes or No. (since 3.4)

OPACITY [Windows and GTK Only]: sets the dialog transparency alpha value. Valid values range from 0 (completely transparent) to 255 (opaque). In Windows must be set before map so the native window would be properly initialized when mapped (since 3.16). (GTK 2.12)

OPACITYIMAGE [Windows and GTK Only]: sets a transparent image as the dialog shape so it is possible to create a non rectangle window. In Windows must be set before map so the native window would be properly initialized when mapped (since 3.16). In GTK the shape works only as a bitmap mask, to view a color image must also use a label. (GTK 2.12) (since 3.12)

TOPMOST [Windows and GTK Only]: puts the dialog always in front of all other dialogs in all applications. Default: NO.

Exclusive Taskbar and Tray/Status Area [Windows and GTK Only]

HIDETASKBAR [Windows and GTK Only] (write-only): Action attribute that when set to "YES", hides the dialog, but does not decrement the visible dialog count, does not call SHOW_CB and does not mark the dialog as hidden inside IUP. It is usually used to hide the dialog and keep the tray icon working without closing the main loop. It has the same effect as setting LOCKLOOP=Yes and normally hiding the dialog. IMPORTANT: when you hide using HIDETASKBAR, you must show using HIDETASKBAR also. Possible values: YES, NO.

TRAY [Windows and GTK Only]: When set to "YES", displays an icon on the system tray. (GTK 2.10)

TRAYIMAGE [Windows and GTK Only]: Name of a IUP image to be used as the tray icon. The Windows SDK recommends that cursors and icons should be implemented as resources rather than created at run time. (GTK 2.10)

TRAYTIP [Windows and GTK Only]: Tray icon's tooltip text. (GTK 2.10)

TRAYTIPMARKUP [GTK Only]: allows the tip string to contains Pango markup commands. Can be "YES" or "NO". Default: "NO". Must be set before setting the TRAYTIP attribute. (GTK 2.16) (since 3.6)

TRAYTIPBALLOON [Windows Only]: The tip window will have the appearance of a cartoon "balloon" with rounded corners and a stem pointing to the item. Default: NO. Must be set before setting the TRAYTIP attribute. (since 3.6)

TRAYTIPBALLOONDELAY [Windows Only]: Time the tip will remain visible. Default is system dependent. The minimum and maximum values are 10000 and 30000 milliseconds. Must be set before setting the TRAYTIP attribute. (since 3.6)

TRAYTIPBALLOONTITLE [Windows Only]: When using the balloon format, the tip can also has a title in a separate area. Must be set before setting the TRAYTIP attribute. (since 3.6)

TRAYTIPBALLOONTITLEICON [Windows Only]: When using the balloon format, the tip can also has a pre-defined icon in the title area. Must be set before setting the TRAYTIP attribute. (since 3.6)

Values can be:
"0" - No icon (default)
"1" - Info icon
"2" - Warning icon
"3" - Error Icon

Exclusive [GTK Only]

DIALOGHINT (creation-only): if enabled sets the window type hint to a dialog hint.

Exclusive [Windows Only]

BRINGFRONT [Windows Only] (write-only): makes the dialog the foreground window. Use "YES" to activate it. Useful for multithreaded applications.

COMPOSITED [Windows Only] (creation only): controls if the window will have an automatic double buffer for all children. Default is "NO". In Windows Vista it is NOT working as expected.

CONTROL [Windows Only] (creation only): Embeds the dialog inside another window.

HELPBUTTON [Windows Only] (creation only): Inserts a help button in the same place of the maximize button. It can only be used for dialogs without the minimize and maximize buttons, and with the menu box. For the next interaction of the user with a control in the dialog, the callback HELP_CB will be called instead of the control defined ACTION callback. Possible values: YES, NO. Default: NO.

MAXIMIZED [Windows Only] (read-only): indicates if the dialog is maximized. Can be YES or NO. (since 3.12)

MINIMIZED [Windows Only] (read-only): indicates if the dialog is minimized. Can be YES or NO. (since 3.15)

TASKBARPROGRESS [Windows Only] (write-only): this functionality enables the use of progress bar on a taskbar button (Windows 7 or earlier version) (Available only for Visual C++ 10 and above). Default: NO (since 3.10).

TASKBARPROGRESSSTATE [Windows Only] (write-only): sets the type and state of the progress indicator displayed on a taskbar button. Possible values: NORMAL (a green bar), PAUSED (a yellow bar), ERROR (a red bar), INDETERMINATE (a green marquee) and NOPROGRESS (no bar). Default: NORMAL (since 3.10).

TASKBARPROGRESSVALUE [Windows Only] (write-only): updates a progress bar hosted in a taskbar button to show the specific percentage completed of the full operation. The value must be between 0 and 100 (since 3.10).

TOOLBOX [Windows Only] (creation only): makes the dialog look like a toolbox with a smaller title bar. It is only valid if the PARENTDIALOG or NATIVEPARENT attribute is also defined. Default: NO.

Exclusive MDI [Windows Only]

--- For the MDI Frame ---

MDIFRAME (creation only) [Windows Only] (non inheritable): Configure this dialog as a MDI frame. Can be YES or NO. Default: NO.

MDIACTIVE [Windows Only] (read-only): Returns the name of the current active MDI child. Use IupGetAttributeHandle to directly retrieve the child handle.

MDIACTIVATE [Windows Only] (write-only): Name of a MDI child window to be activated. If value is "NEXT" will activate the next window after the current active window. If value is "PREVIOUS" will activate the previous one.

MDIARRANGE [Windows Only] (write-only): Action to arrange MDI child windows. Possible values: TILEHORIZONTAL, TILEVERTICAL, CASCADE and ICON (arrange the minimized icons).

MDICLOSEALL [Windows Only] (write-only): Action to close and destroy all MDI child windows. The CLOSE_CB callback will be called for each child.

IMPORTANT: When a MDI child window is closed it is automatically destroyed. The application can override this returning IUP_IGNORE in CLOSE_CB.

MDINEXT [Windows Only] (read-only): Returns the name of the next available MDI child. Use IupGetAttributeHandle to directly retrieve the child handle. Must use MDIACTIVE to retrieve the first child. If the application is going to destroy the child retrieve the next child before destroying the current.

--- For the MDI Client (a IupCanvas) **---**

MDICLIENT (creation only) [Windows Only] (non inheritable): Configure the canvas as a MDI client. Can be YES or NO. No callbacks will be called. This canvas will be used internally only by the MDI Frame and its MDI Children. The MDI frame must have one and only one MDI client. Default: NO.

MDIMENU (creation only) [Windows Only]: Name of a IupMenu to be used as the Window list of a MDI frame. The system will automatically add the list of MDI child windows there.

--- For the MDI Children ---

MDICHILD (creation only) [Windows Only]: Configure this dialog to be a MDI child. Can be YES or NO. The PARENTDIALOG attribute must also be defined. Each MDI child is automatically named if it does not have one. Default: NO.

Callbacks

CLOSE_CB: Called right before the dialog is closed.

COPYDATA_CB [Windows Only]: Called at the first instance, when a second instance is running. Must set the global attribute SINGLEINSTANCE to be called. (since 3.2)

```
int function(Ihandle *ih, char* cmdLine, int size); [in C]
elem:copydata_cb(cmdLine: string, size: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
cmdLine: command line of the second instance.
size: size of the command line string including the null character.

DROPFILES_CB [Windows and GTK Only]: Action generated when one or more files are dropped in the dialog.

MDIACTIVATE_CB [Windows Only]: Called when a MDI child window is activated. Only the MDI child receive this message. It is not called when the child is shown for the first time.

```
int function(Ihandle *ih); [in C]
elem:mdiactivate_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

MOVE_CB [Windows and GTK Only]: Called after the dialog was moved on screen. The coordinates are the same as the SCREENPOSITION attribute. (since 3.0)

```
int function(Ihandle *ih, int x, int y); [in C]
elem:move_cb(x, y: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
x, **y**: coordinates of the new position.

RESIZE_CB: Action generated when the dialog size is changed. If returns IUP_IGNORE the dialog layout is NOT recalculated. (since 3.0)

SHOW_CB: Called right after the dialog is showed, hidden, maximized, minimized or restored from minimized/maximized.

TRAYCLICK_CB [Windows and GTK Only]: Called right after the mouse button is pressed or released over the tray icon. (GTK 2.10)

```
int function(Ihandle *ih, int but, int pressed, int dclick); [in C]
elem:trayclick_cb(but, pressed, dclick: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
but: identifies the activated mouse button. Can be: 1, 2 or 3. Note that this is different from the BUTTON_CB canvas callback definition. GTK does not get button=2 messages.
pressed: indicates the state of the button. Always 1 in GTK.
dclick: indicates a double click. In GTK double click is simulated.

Returns: IUP_CLOSE will be processed.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Drag & Drop attributes and callbacks are supported.

Notes

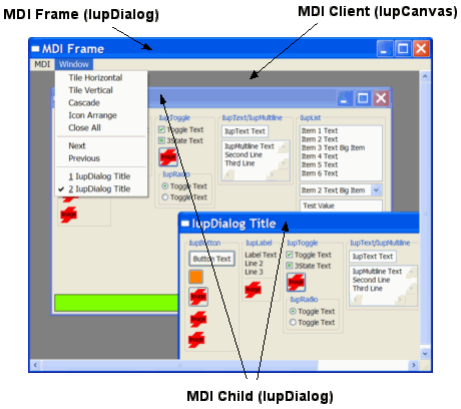
Do not associate an **IupDialog** with the native "dialog" nomenclature in Windows, GTK or Motif. **IupDialog** use native standard windows in all drivers.

Except for the menu, all other elements must be inside a dialog to interact with the user. Therefore, an interface element will only be visible if its dialog is also visible.

The order of callback calling is system dependent. For instance, the RESIZE_CB and the SHOW_CB are called in a different order in Win32 and in X-Windows when the dialog is shown for the first time.

Windows MDI

The MDI support is composed of 3 components: the MDI frame window (IupDialog), the MDI client window (IupCanvas) and the MDI children (IupDialog). Although the MDI client is a IupCanvas it is not used directly by the application, but it must be created and included in the dialog that will be the MDI frame, other controls can also be available in the same dialog, like buttons and other canvases composing toolbars and status area. The following picture illustrates the e components:



Examples

Very simple dialog with a label and a button. The application is closed when the button is pressed.

```
#include <iup.h>

int quit_cb(void)
{
  return IUP_CLOSE;
}

int main(int argc, char* argv[])
{
  Ihandle *dialog, *quit_bt, *vbox;

  IupOpen(&argc, &argv);

  /* Creating the button */
  quit_bt = IupButton("Quit", 0);
  IupSetCallback(quit_bt, "ACTION", (Icallback)quit_cb);

  /* the container with a label and the button */
  vbox = IupVbox(
           IupSetAttributes(IupLabel("Very Long Text Label"), "EXPAND=YES, ALIGNMENT=ACENTER"),
           quit_bt,
           0);
  IupSetAttribute(vbox, "MARGIN", "10x10");
  IupSetAttribute(vbox, "GAP", "5");

  /* Creating the dialog */
  dialog = IupDialog(vbox);
  IupSetAttribute(dialog, "TITLE", "Dialog Title");
  IupSetAttributeHandle(dialog, "DEFAULTESC", quit_bt);

  IupShow(dialog);

  IupMainLoop();

  IupDestroy(dialog);
  IupClose();

  return 0;
}
```

Browse for Example Files

See Also

IupFileDlg, IupMessageDlg, IupDestroy, IupShowXY, IupShow, IupPopup

CURSOR (non inheritable)

Defines the element's cursor.

Value

Name of a cursor.

It will check first for the following predefined names:

		Name
		"NONE" or "NULL"
		"ARROW"
		"BUSY"
		"CROSS"

		"HAND"
		"HELP"
		"MOVE"
		"PEN" (*)
		"RESIZE_N"
		"RESIZE_S"
		"RESIZE_NS"
		"RESIZE_W"
		"RESIZE_E"
		"RESIZE_WE"
		"RESIZE_NE"
		"RESIZE_SW"
		"RESIZE_NW"
		"RESIZE_SE"
		"TEXT"
	----	"APPSTARTING" (Windows Only)
	----	"NO" (Windows Only)
		"UPARROW"

Default: "ARROW"

(*) To use this cursor on Windows, the **iup.rc** file, provided with IUP, must be linked with the application (except when using the IUP DLL).

The Windows SDK recommends that cursors and icons should be implemented as resources rather than created at run time.

The GTK cursors have the same appearance of the X-Windows cursors. Althought GTK cursors can have more than 2 colors depending on the X-Server.

If it is not a pre-defined name, then will check for other system cursors. In Windows the value will be used to load a cursor form the application resources. In Motif the value will be used as a X-Windows cursor number, see definitions in the X11 header "cursorfont.h". In GTK the value will be used as a cursor name, see the GDK documentation on Cursors.

If no system cursors were found then the value will be used to try to find an IUP image with the same name. Use **IupSetHandle** to define a name for an **IupImage**. But the image will need an extra attribute and some specific characteristics, see notes below.

Notes

For an image to represent a cursor, it should has the attribute "**HOTSPOT**" to define the cursor hotspot (place where the mouse click is actually effective). The default value is "0:0".

Usually only color indices 0, 1 and 2 can be used in a cursor, where 0 will be transparent (must be "BGCOLOR"). The RGB colors corresponding to indices 1 and 2 are defined just as in regular images. In Windows and GTK the cursor can have more than 2 colors. Cursor sizes are usually less than or equal to 32x32.

The cursor will only change when the interface system regains control or when IupFlush is called.

The Windows SDK recommends that cursors and icons should be implemented as resources rather than created at run time.

When the cursor image is no longer necessary, it must be destroyed through function IupDestroy. Attention: the cursor cannot be in use when it is destroyed.

Affects

IupDialog, IupCanvas

See Also

IupImage

ICON

Dialog's icon. This icon will be used when the dialog is minimized.

Value

Name of a IUP image.

Default: NULL

Notes

Icon sizes are usually less than or equal to 32x32.

The Windows SDK recommends that cursors and icons should be implemented as resources rather than created at run time. We suggest using an icon with at least 3 images: 16x16 32bpp, 32x32 32 bpp and 48x48 32 bpp.

On Motif, it only works with some window managers, like *mwm* and *gnome*. Icon colors can have the BGCOLOR values, but it works better if it is at index 0.

Use IupSetHandle or IupSetAttributeHandle to associate an image to a name.

Affects

IupDialog

See Also

IupImage

PARENTDIALOG

The parent dialog of a dialog.

Value

Name of a dialog to be used as parent.

Default: NULL.

Notes

This dialog will be always in front of the parent dialog. If the parent is minimized, this dialog is automatically minimized. The parent dialog must be mapped before mapping the child dialog.

If PARENTDIALOG is not defined then the NATIVEPARENT attribute is consulted. This one must be a native handle of an existing dialog.

Use IupSetHandle or IupSetAttributeHandle to associate a dialog to a name.

IMPORTANT: When the parent is destroyed the child dialog is also destroyed, BUT the CLOSE_CB callback of the child dialog is NOT called. The application must take care of destroying the child dialogs before destroying the parent. This is usually done when CLOSE_CB of the parent dialog is called.

Affects

IupDialog

SHRINK

If this attribute is defined, the elements inside the dialog will try to adjust their sizes even when the dialog's size is smaller than its natural size.

See the Layout Guide for more details on sizes.

Value

"YES" or "NO".

Default: "NO".

Notes

When the user changes the size of the dialog, the elements are automatically re-distributed inside the dialog. Some elements even have their size changed if the EXPAND attribute is active. When this size is smaller than a minimum limit in which all elements still fit the dialog, the elements' distribution is no longer modified. Actually, the virtual size of the dialog remains larger than its actual size on the screen, and some elements to the right and bottom are hidden by the borders of the dialog.

The SHRINK attribute offers an alternative to this behavior. It makes the elements continue to rearrange, even if they must overlapi

The results of this new rearrangement may vary according to the elements' distribution on the dialog.

Shrink will be effective only for containers. For regular elements the current size will be set for a smaller value only if EXPAND is set.

See the Layout Guide for more details on sizes.

Affects

IupDialog

CONTROL

Windows only. Whether the dialog is embedded inside the parent window or has a window of its own.

Value

YES or NO. If the value is YES, the dialog will appear embedded inside its parent window (you must set a parent, either with PARENTDIALOG or NATIVEPARENT, or this setting will be ignored). If the value is NO, the dialog will have its own window.

Notes

This is useful for implementing ActiveX controls, with the help of the LuaCOM library. ActiveX controls run embedded inside another window. LuaCOM will send a window creation event the the control, passing a handle to the parent window and the size of the control. You can use this to set the dialog's NATIVEPARENT and RASTERSIZE attributes. See the LuaCOM documentation for more information.

Affects

IupDialog

See Also

NATIVEPARENT, **PARENTDIALOG**, **RASTERSIZE**

CLOSE_CB

Called just before a dialog is closed when the user clicks the close button of the title bar or an equivalent action.

Callback

```
int function(Ihandle *ih); [in C]
ih:close_cb() -> (ret: number) [in Lua]
```

ih: identifies the element that activated the event.

Returns: if IUP_IGNORE, it prevents the dialog from being closed. If you destroy the dialog in this callback, you must return IUP_IGNORE. IUP_CLOSE will be processed.

Affects

IupDialog

DROPFILES_CB

Action called when a file is "dropped" into the control. When several files are dropped at once, the callback is called several times, once for each file.

If defined after the element is mapped then the attribute DROPFILESTARGET must be set to YES.

[Windows and GTK Only] (GTK 2.6)

Callback

```
int function(Ihandle *ih, const char* filename, int num, int x, int y); [in C]
ih:dropfiles_cb(filename: string; num, x, y: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
filename: Name of the dropped file.
num: Number index of the dropped file. If several files are dropped, **num** is the index of the dropped file starting from "total-1" to "0".
x: X coordinate of the point where the user released the mouse button.
y: Y coordinate of the point where the user released the mouse button.

Returns: If IUP_IGNORE is returned the callback will NOT be called for the next dropped files, and the processing of dropped files will be interrupted.

Affects

IupDialog, IupCanvas, IupGLCanvas, IupText, IupList

RESIZE_CB

Action generated when the canvas or dialog size is changed.

Callback

```
int function(Ihandle *ih, int width, int height); [in C]
ih:resize_cb(width, height: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
width: the width of the internal element size in pixels not considering the decorations (client size)
height: the height of the internal element size in pixels not considering the decorations (client size)

Notes

For the dialog, this action is also generated when the dialog is mapped, after the map and before the show.

When XAUTOHIDE=Yes or YAUTOHIDE=Yes, if the canvas scrollbar is hidden/shown after changing the DX or DY attributes from inside the callback, the size of the drawing area will immediately change, so the parameters **with** and **height** will be invalid. To update the parameters consult the DRAWSIZE attribute. Also activate the drawing toolkit only after updating the DX or DY attributes.

Affects

IupCanvas, IupGLCanvas, IupDialog

SHOW_CB

Called right after the dialog is showed, hidden, maximized, minimized or restored from minimized/maximized. This callback is called when those actions were performed by the user or programmatically by the application.

Callback

```
int function(Ihandle *ih, int state); [in C]
ih:show_cb(state: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
state: indicates which of the following situations generated the event:

IUP_HIDE (since 3.0)
IUP_SHOW
IUP_RESTORE (was minimized or maximized)
IUP_MINIMIZE
IUP_MAXIMIZE (since 3.0) (not received in Motif when activated from the maximize button)

Returns: IUP_CLOSE will be processed.

Affects

IupDialog

IupPopup

Shows a dialog or menu and restricts user interaction only to the specified element. It is equivalent of creating a Modal dialog is some toolkits.

If another dialog is shown after **IupPopup** using **IupShow**, then its interaction will not be inhibited. Every **IupPopup** call creates a new popup level that inhibits all previous dialogs interactions, but does not disable new ones (even if they were disabled by the Popup, calling **IupShow** will re-enable the dialog because it will change its popup level). IMPORTANT: The popup levels must be closed in the reverse order they were created or unpredictable results will occur.

For a dialog this function will only return the control to the application after a callback returns IUP_CLOSE, **IupExitLoop** is called, or when the popup dialog is hidden, for example using **IupHide**. For a menu it returns automatically after a menu item is selected. IMPORTANT: If a menu item callback returns IUP_CLOSE, it will also ends the current popup level dialog.

Parameters/Return

```
int IupPopup(Ihandle *ih, int x, int y); [in C]
iup.Popup(ih: ihandle[, x, y: number]) -> (ret: number) [in Lua]
or ih:popup([x, y: number]) -> (ret: number) [in Lua]
```

ih: Identifier of a dialog or a menu.
x: horizontal position of the top left corner of the window or menu, relative to the origin of the main screen. The following definitions can also be used:

- IUP_LEFT: Positions the element on the left corner of the main screen
- IUP_CENTER: Centers the element on the main screen
- IUP_RIGHT: Positions the element on the right corner of the main screen
- IUP_MOUSEPOS: Positions the element on the mouse cursor
- IUP_CENTERPARENT: Horizontally centralizes the dialog relative to its parent. Not valid for menus. (Since 3.0)
- IUP_CURRENT: use the current position of the dialog. This is the default value in Lua if the parameter is not defined. Not valid for menus. (Since 3.0)

y: vertical position of the top left corner of the window or menu, relative to the origin of the main screen. The following definitions can also be used:

- IUP_TOP: Positions the element on the top of the main screen
- IUP_CENTER: Vertically centers the element on the main screen
- IUP_BOTTOM: Positions the element on the base of the main screen
- IUP_MOUSEPOS: Positions the element on the mouse cursor
- IUP_CENTERPARENT: Vertically centralizes the dialog relative to its parent. Not valid for menus. (Since 3.0)
- IUP_CURRENT: use the current position of the dialog. This is the default value in Lua if the parameter is not defined. Not valid for menus. (Since 3.0)

Returns: IUP_NOERROR if successful. Returns IUP_INVALID if not a dialog or menu. If there was an error returns IUP_ERROR..

Notes

It will call **IupMap** for the element.

The **x** and **y** values are the same as returned by the SCREENPOSITION attribute.

IUP_MOUSEPOS position is the same as returned by the CURSORPOS global attribute.

See the PLACEMENT attribute for other position and show options.

When IUP_CENTERPARENT is used but PARENTDIALOG is not defined then it is replaced by IUP_CENTER.

When IUP_CURRENT is used at the first time the dialog is shown then it will be replaced by IUP_CENTERPARENT.

The main screen area does not include additional monitors.

IupPopup works just like **IupShowXY**, but it inhibits interaction with other dialogs that are visible and interrupts the processing until IUP_CLOSE is returned in a callback, **IupExitLoop** is called, or the popup dialog is hidden. This is now a modal dialog. Although it interrupts the processing, it does not destroy the dialog when it ends. To destroy the dialog, **IupDestroy** must be called.

The MODAL attribute of the dialog will be changed internally to return Yes.

In GTK and Motif the inactive dialogs will still be moveable, resizable and changeable their Z-order. Although their contents will be inactive, keyboard will be disabled, and they can not be closed from the close box.

When called for an already visible dialog (modal or not) it will update its position and PLACEMENT. If the already visible dialog is not modal then it will became modal and processing will be interrupted as a regular **IupPopup** (since 3.16). If the already visible dialog is modal then the function returns and it will NOT interrupt processing, the dialog still will remains MODAL. In other words, calling **IupPopup** a second time will just update the dialog position and it will not interrupt processing, and calling **IupPopup** for a dialog shown with **IupShowXY** will turn it a modal dialog.

See Also

IupShowXY, IupShow, IupHide, IupMap, IupDialog

IupShow

Displays a dialog in the current position, or changes a control VISIBLE attribute.

Parameters/Return

```
int IupShow(Ihandle *ih); [in C]
iup.Show(ih: ihandle) -> (ret: number) [in Lua]
or ih:show() -> (ret: number) [in IupLua]
```

ih: identifier of the interface element.

Returns: IUP_NOERROR if successful. If there was an error returns IUP_ERROR.

Notes

For dialogs it is equivalent to call **IupShowXY** using IUP_CURRENT. See IupShowXY for more details.

For other controls, to call **IupShow** is the same as setting VISIBLE=YES.

See Also

IupShowXY, IupHide, IupPopup, IupMap

IupShowXY

Displays a dialog in a given position on the screen.

Parameters/Return

```
int IupShowXY(Ihandle *ih, int x, int y); [in C]
iup.ShowXY(ih: ihandle[, x, y: number]) -> (ret: number) [in Lua]
or ih:showxy([x, y: number]) -> (ret: number) [in Lua]
```

ih: identifier of the dialog.
x: horizontal position of the top left corner of the window, relative to the origin of the main screen. The following definitions can also be used:

- IUP_LEFT: Positions the dialog on the left corner of the main screen
- IUP_CENTER: Horizontally centralizes the dialog on the main screen
- IUP_RIGHT: Positions the dialog on the right corner of the main screen
- IUP_MOUSEPOS: Positions the dialog on the mouse position
- IUP_CENTERPARENT: Horizontally centralizes the dialog relative to its parent (Since 3.0)
- IUP_CURRENT: use the current position of the dialog. This is the default value in Lua if the parameter is not defined. (Since 3.0)

y: vertical position of the top left corner of the window, relative to the origin of the main screen. The following definitions can also be used:

- IUP_TOP: Positions the dialog on the top of the main screen
- IUP_CENTER: Vertically centralizes the dialog on the main screen
- IUP_BOTTOM: Positions the dialog on the base of the main screen
- IUP_MOUSEPOS: Positions the dialog on the mouse position

- IUP_CENTERPARENT: Vertically centralizes the dialog relative to its parent (Since 3.0)
- IUP_CURRENT: use the current position of the dialog. This is the default value in Lua if the parameter is not defined.(Since 3.0)

Returns: IUP_NOERROR if successful. Returns IUP_INVALID if not a dialog. If there was an error returns IUP_ERROR.

Notes

Will call **IupMap** for the element.

x and **y** positions are the same as returned by the SCREENPOSITION attribute.

IUP_MOUSEPOS position is the same as returned by the CURSORPOS global attribute.

See the PLACEMENT attribute for other position and show options.

When IUP_CENTERPARENT is used but PARENTDIALOG is not defined then it is replaced by IUP_CENTER.

When IUP_CURRENT is used at the first time the dialog is shown then it will be replaced by IUP_CENTERPARENT.

The main screen area does not include additional monitors.

When called for an already visible dialog (MODAL or not) it will update its position and PLACEMENT. If the already visible dialog is not modal but it was disabled by one (a call to **IupPopup** for another dialog after this one was shown) then it will be now enabled for interaction.

See Also

IupShow, IupHide, IupPopup, IupMap, IupDialog

IupHide

Hides an interface element. This function has the same effect as attributing value "NO" to the interface element's VISIBLE attribute.

Parameters/Return

```
int IupHide(Ihandle *ih); [in C]
iup.Hide(ih: ihandle) -> (ret: number) [in Lua]
or ih:hide() -> (ret: number) [in Lua]
```

ih: Identifier of the interface element.

Returns: IUP_NOERROR always.

Notes

Once a dialog is hidden, either by means of **IupHide** or by changing the VISIBLE attribute or by means of a click in the window close button, the elements inside this dialog are not destroyed, so that you can show the dialog again. To destroy dialogs, the **IupDestroy** function must be called.

See Also

IupShowXY, IupShow, IupPopup, IupDestroy.

Browse for Example Files

See Also

IupMessage, IupScanf, IupListDialog, IupAlarm, IupGetFile, IupPopup

See Also

IupMessage, IupScanf, IupListDialog, IupAlarm, IupGetFile, IupPopup

See Also

IupMessageDlg, IupFileDlg, IupPopup

See Also

IupMessageDlg, IupFileDlg, IupPopup

See Also

IupProgressBar, IupDialog

IupAlarm

Shows a modal dialog containing a message and up to three buttons.

Creation and Show

```
int IupAlarm(const char *t, const char *m, const char *b1, const char *b2, const char *b3); [in C]
iup.Alarm(t, m, b1[, b2, b3]: string) -> (button: number) [in Lua]
```

t: Dialog's title
m: Message
b1: Text of the first button
b2: Text of the second button (optional)
b3: Text of the third button (optional)

Returns: the number of the **button** selected by the user (1, 2 or 3) , or 0 if failed. It fails only if b1 is not defined.

Notes

This function shows a dialog centralized on the screen, with the message and the buttons. The '\n' character can be added to the message to indicate line change.

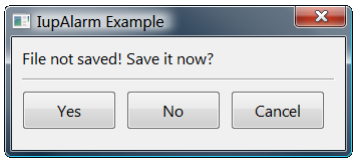
A button is not shown if its parameter is NULL. This is valid only for **b2** and **b3**.

Button 1 is set as the "DEFAULTENTER" and "DEFAULTESC". If Button 2 exists it is set as the "DEFAULTESC". If Button 3 exists it is set as the "DEFAULTESC".

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

Examples

Browse for Example Files



See Also

IupMessage, IupScanf, IupListDialog, IupGetFile.

IupGetFile

Shows a modal dialog of the native interface system to select a filename. Uses the **IupFileDlg** element.

Creation and Show

```
int IupGetFile(char *filename); [in C]
iup.GetFile(filename: string) -> (filename: string, status: number) [in Lua]
```

filename: This parameter is used as an input value to define the default filter and directory. Example: "../docs/*.txt". As an output value, it is used to contain the filename entered by the user.

Returns: a **status** code, whose values can be:

"1": New file.
"0": Normal, existing file.
"-1": Operation cancelled.

Notes

The function does not allocate memory space to store the complete filename entered by the user. Therefore, the filename parameter must be large enough to contain the directory and file names. The string is limited to 4096 characters.

The function will reuse the directory from one call to another, so in the next call will open in the directory of the last selected file.

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

For a more controlled dialog use directly the IupFileDlg element.

Examples

Browse for Example Files

See Also

IupFileDlg, IupMessage, IupScanf, IupListDialog, IupAlarm, IupSetLanguage.

IupGetColor

Shows a modal dialog which allows the user to select a color. Based on IupColorDlg.

Creation and Show

```
int IupGetColor(int x, int y, unsigned char *r, unsigned char *g, unsigned char *b); [in C]
iup.GetColor(x, y[, r, g, b: number]) -> (r, g, b: number)  [in Lua]
```

x, y: x, y values of the **IupPopup** function.
r, g, b: Pointers to variables that will receive the color selected by the user if the OK button is pressed. The value in the variables at the moment the function is called defines the color being selected when the dialog is shown. If the OK button is not pressed, the r, g and b values are not changed. These values cannot be NULL in C, in Lua they are optional and used for initialization only.

Returns: in C a code 1 if the OK button is pressed, or 0 otherwise. In Lua the code is not returned, instead the r,g,b values are returned or nil otherwise.

Notes

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

Examples

Browse for Example Files

See Also

IupMessage, IupScanf, IupListDialog, IupAlarm, IupGetFile.

IupGetParam

Shows a modal dialog for capturing parameter values using several types of controls.

Creation and Show

```
int IupGetParam(const char* title, Iparamcb action, void* user_data, const char* format,...); [in C]
int IupGetParamv(const char* title, Iparamcb action, void* user_data, const char* format, int param_count, int param_extra, void** param_data); [in C]
iup.GetParam(title: string, action: function, format: string,...) -> (status: boolean, ...) [in Lua]
```

title: dialog title.
action: user callback to be called whenever a parameter value was changed, and when the user pressed the OK button. It can be NULL.
user_data: user pointer passed to the user callback.
format: string describing all the parameters, see Notes.
...: list of variables address with initial values for the parameters.
param_count: number of regular parameters in the array.
param_extra: number of extra parameters in the array (separator lines and button names).
param_data: array of variables address with initial values for the parameters.

Returns: a **status** code 1 if the button 1 was pressed, 0 if the button 2 was pressed or if an error occurred.

The function will abort if there are errors in the format string as in the number of the expected parameters. In Lua, the values are returned by the function in the same order they were passed. The Lua type of each parameter is the equivalent C type (boolean is integer), except for the **status** code that it is a boolean.

Callbacks

```
int function(Ihandle* dialog, int param_index, void* user_data); [in C]
luafunction(dialog: ihandle, param_index: number) -> (ret: number) [in Lua]
```

dialog: dialog handle
param_index: current parameter being changed. Can have negative values to indicate specific situations:
IUP_GETPARAM_BUTTON1 (-1) = if the user pressed the button 1;
IUP_GETPARAM_INIT (-2) = after the dialog is **mapped** and just before it is shown. Not called for **IupParamBox**;
IUP_GETPARAM_BUTTON2 (-3) = if the user pressed the button 2;
IUP_GETPARAM_BUTTON3 (-4) = if the user pressed the button 3, if any;
IUP_GETPARAM_CLOSE (-5) = if the user clicked on the dialog close button. Not called for **IupParamBox**; (since 3.13)
user_data: a user pointer that is passed in the function call.

Returns: You can reject the change or the button action by returning 0 in the callback, otherwise you must return 1. By default button 1 and button2 will close the dialog.

Internally the callback is stored as a regular callback with the "PARAM_CB" name.

You should not programmatically change the current parameter value during the callback. On the other hand you can freely change the value of other parameters.

Use the dialog attribute "PARAMn" to get the parameter "Ihandle*", where "n" is the parameter index in the order they are specified starting at 0, but separators and button names are not counted. Notice that this is not the actual control, use the parameter attribute "CONTROL" to get the actual control. For example:

```
Ihandle* param2 = (Ihandle*)IupGetAttribute(dialog, "PARAM2");
int value2 = IupGetInt(param2, IUP_VALUE);

Ihandle* param5 = (Ihandle*)IupGetAttribute(dialog, "PARAM5");
Ihandle* ctrl5 = (Ihandle*)IupGetAttribute(param5, "CONTROL");

if (value2 == 0)
{
  IupSetAttribute(param5, IUP_VALUE, "New Value");
  IupSetAttribute(ctrl5, IUP_VALUE, "New Value");
}
```

Since parameters are user controls and not real controls, you must update the control value and the parameter value.

Be aware that programmatically changes are not filtered. The valuator, when available, can be retrieved using the parameter attribute "AUXCONTROL". The valuator is not automatically updated when the text box is changed programmatically. The parameter label is also available using the parameter attribute "LABEL".

Attributes (inside the callback)

For the dialog:

"PARAMn" - returns an IUP Ihandle* representing the n^{th} parameter, indexed by the declaration order, not counting separators or button names.
"BUTTON1" - returns an IUP Ihandle*, the button 1.
"BUTTON2" - returns an IUP Ihandle*, the button 2.
"BUTTON3" - returns an IUP Ihandle*, the button 3.

For a parameter:

"LABEL" - returns an IUP Ihandle*, the label associated with the parameter.
"CONTROL" - returns an IUP Ihandle*, the real control associated with the parameter.
"AUXCONTROL" - returns an IUP Ihandle*, the auxiliary control associated with the parameter (only for Valuators).
"INDEX" - returns an integer value associated with the parameter index. **IupGetInt** can also be used.
"VALUE" - returns the value of the parameter. **IupGetFloat** and **IupGetInt** can also be used. For the current parameter inside the callback contains the new value that will be applied to the control, to get the old value use the VALUE attribute for the CONTROL returned Ihandle*.

In Lua, to retrieve a parameter you must use the following function:

```
iup.GetParamParam(dialog: ihandle, param_index: number)-> (param: ihandle) [in Lua]
```

dialog: Identifier of the dialog.
param_index: parameter to be retrieved.

In Lua, to retrieve a parameter control, auxiliary control or button you must use the following function:

```
iup.GetParamHandle(param: ihandle, name: string)-> (control: ihandle) [in Lua] (since 3.16)
```

param: Identifier of the parameter.
name: name of the parameter.

Notes

The format string must have the following format, notice the "\n" at the end

"***text***%***x*[*extra*]{**tip**}\n", where:

text is a descriptive text, to be placed to the left of the entry field in a label. It can contains any string, but to contain a '%' must use two characters "%%" to avoid conflict with the type separator (since 3.6). If it is preceded by n '\t' characters then the parameter will be indented by the same number (since 3.13).

x is the type of the parameter. The valid options are:

b = boolean (shows a True/False toggle, use "int" in C)
i = integer (shows a integer number filtered text box, use "int" in C)
r = real (shows a real number filtered text box, use "float" in C)
R = same as **r** but using "double" in C (since 3.11.1)
a = angle in degrees (shows a real number filtered text box and a dial [if **IupControlsOpen** were called], use "float" in C)
A = same as **a** but using "double" in C (since 3.11.1)
s = string (shows a text box, use "char*" in C, it must have room enough for your string)
m = multiline string (shows a multiline text box, use "char*" in C, it must have room enough for your string)
l = list (shows a dropdown list box, use "int" in C for the zero based item index selected)
o = list (shows a list of toggles inside a radio, use "int" in C for the zero based item index selected) (since 3.3)
t = separator (shows a horizontal line separator label, in this case text can be an empty string, not included in parameter count)
d = string, but the interface uses a IupDatePick element to select a date (since 3.17)
f = string (same as **s**, but also show a button to open a file selection dialog box)
c = string (same as **s**, but also show a color button to open a color selection dialog box)
n = string (same as **s**, but also show a font button to open a font selection dialog box) (since 3.3)
h = Ihandle* (a control handle that will be managed by the application, it will be placed after the parameters and before the buttons.) (since 3.9)
u = buttons titles (allow to redefine the default button titles (OK and Cancel), and to add a third button, use [button1,button2,button3] as extra data, can omit one of them, it will use the default name, not included in parameter count) (since 3.1)

extra is one or more additional options for the given type

[min,max,step] are optional limits for integer and real types. The **max** and **step** values can be omitted. When **min** and **max** are specified a valuator will also be added to change the value. To specify **step**, **max** must be also specified. **step** is the size of the increment.
[false,true] are optional strings for boolean types to be displayed after the toggle. The strings can not have commas '**,**', nor brackets '**[**' or '**]**'.
mask is an optional mask for the string and multiline types. The dialog uses the MASK attribute internally. In this case we do no use the brackets '**[**' and '**]**' to avoid conflict with the specified mask.
|item0|item1|item2,...| are the items of the list. At least one item must exist. Again the brackets are not used to increase the possibilities for the strings, instead you must use '**|**'. Items index are zero based start.
[dialogtype|filter|directory|nochangedir|nooverwriteprompt] are the respective attribute values passed to the IupFileDlg control when activated. All '**|**' must exist, but you can let empty values to use the default values. No mask can be set.

tip is a string that is displayed in a TIP for the main control of the parameter. (since 3.0)

The number of lines in the format string (number of '\n') will determine the number of required parameters. But separators will not count as parameters. There is no maximum number of parameters (since 3.13).

Since the **tip** string can not contain a '\n' because of the param terminator, the '\r' character can be used to break lines in the TIP. It will be internally converted to '\n' before actually setting the TIP. (since 3.17)

A integer parameter always has a spin attached to the text to increment and decrement the value. A real parameter only has a spin in a full interval is defined (min and max), in this case the default step is (max-min)/20. When the callback is called because a spin was activated then the attribute "**SPINNING**" of the dialog will be defined to a non NULL and non zero value.

The default precision for real value display is given by the global attribute DEFAULTPRECISION. But inside the callback the application can set the param attribute "**PRECISION**" to use another value. It will work only during interactive changes. The decimal symbol will used the DEFAULTDECIMALSYMBOL global attribute. (since 3.13)

The function does not allocate memory space to store the text entered by the user. Therefore, the string parameter must be large enough to contain the user input. If you want to set a maximum size for the string you can set the param attribute **MAXSTR**, inside the callback when param_index=IUP_GETPARAM_INIT (since 3.6). Its default value is 10240 for multiline strings, 4096 for file names, and 512 for other strings.

There is no extra parameters for the color string. The mask is automatically set to capture 3 or 4 unsigned integers from 0 to 255 (R G B) or (R G B A) (alpha is optional).

The date extra parameters are simply IupDatePick attributes in a single string for **IupSetAttributes** usage. (since 3.17)

The dialog is resizable if it contains a string, a multiline string or a number with a valuator. All the multiline strings will increase size equally in both directions.

When the "s" type is used the size can be controlled using the VISIBLECOLUMNS attribute at the param element. (since 3.16)

The dialog uses a global attribute called "**PARENTDIALOG**" as the parent dialog if it is defined. It also uses a global attribute called "**ICON**" as the dialog icon if it is defined.

Utility Functions (since 3.13)

Both functions bellow are used internally in **IupGetParam**. But they can be used to integrate the **IupGetParam** contents in other dialogs.

```
Ihandle* IupParamf(const char* format); [in C]
iup.Paramf(format: string) -> elem: ihandle [in Lua]
```

Creates an **IupUser** element to be used in the **IupParamBox** bellow. Each parameter format follows the same specifications as the **IupGetParam** function, including the line feed.

The **IupUser** element can be directly used if the internal attributes are set. Here are the respective attributes according to the **IupGetParam** format specification:

TITLE: text of the parameter, used as label.
INDENT: number of indentation levels.

TYPE: can be BOOLEAN, LIST, OPTIONS, REAL, STRING, INTEGER, DATE, FILE, COLOR, SEPARATOR, BUTTONNAMES and HANDLE. And describe the type of the parameter.
DATATYPE: can be INT (int), FLOAT (float), DOUBLE (double), STRING (char*), HANDLE (Ihandle*) or NONE (when buttons and separators are used). And describe the C data type that must be passed to **IupGetParam** to initialize and receive parameter values.
MULTILINE: can be Yes or No. Defines if the edit box can have more than one line.
ANGLE: can be Yes or No. defines if the REAL type is an angle.

TRUE, **FALSE**: boolean names.
INTERVAL (Yes/No), **MIN**, **MAX**, **STEP**, **PARTIAL** (Yes/No): optional limits for integer and real types.
DIALOGTYPE, **FILTER**, **DIRECTORY**, **NOCHANGEDIR**, **NOOVERWRITEPROMPT**: used for the FILE parameter dialog. See IupFileDlg.
BUTTON1, **BUTTON2**, **BUTTON3**: button titles. Default is "OK/Cancel/Help" for regular **IupGetParam**, and "Apply/Reset/Help" for **IupParamBox**.
0, 1, 2, 3, ... : list items.
MASK: mask for the edit box input.

TIP: text of the tip.

VALUE: the parameter value.
STATUS: set to 1 when button1 is activated, and set to 0 when button2 is activated or the dialog close button is clicked.

```
Ihandle* IupParamBox(Ihandle* parent, Ihandle** params, int count); [in C]
iup.ParamBox(parent: ihandle, params: table of ihandle) -> elem: ihandle [in Lua]
```

Creates the **IupGetParam** dialog contents with the array of parameters. This includes the button box at the bottom.

The buttons of the button box can be retrieved using the **BUTTON1**, **BUTTON2**, **BUTTON3** attribute at the button box. They will directly return the control handle not the button titles as in a **IupParamf**.

The **PARAM_CB** callback will receive the box handle instead of the **IupGetParam** dialog handle. Buttons 1 and 2 will still close the dialog as in **IupGetParam** (by returning IUP_CLOSE internally), so to change that behavior return 0 in the callback. The **USERDATA** attribute will hold the user data passed to the callback.

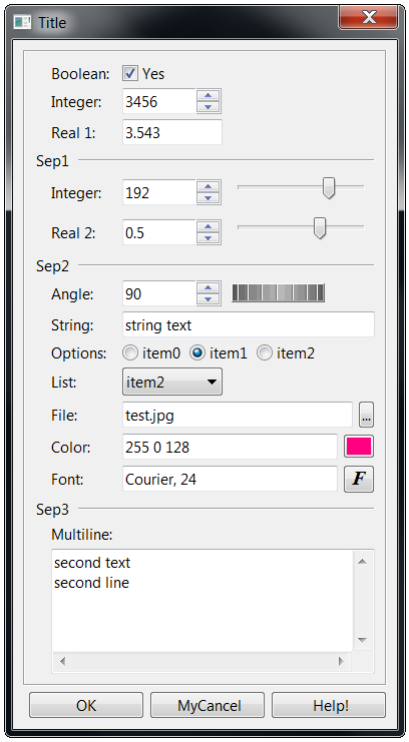
The **PARAMCOUNT** attribute contains the number of parameters not counting separators and button names.

Examples

Browse for Example Files

Here is an example showing many of the possible parameters. We show only one for each type, but you can have as many parameters of the same type you want.

```
int pboolean = 1;
int pinteger = 3456;
float preal = 3.543f;
int pinteger2 = 192;
float preal2 = 0.5f;
float pangle = 90;
char pstring[100] = "string text";
char pfont[100] = "Courier, 24";
char pcolor[100] = "255 0 128";
int plist = 2, poptions = 1;
char pstring2[200] = "second text\nsecond line";
char file_name[500] = "test.jpg";

if (!IupGetParam("Title", param_action, 0,
                 "Bt %u[, MyCancel, Help!]\n"
                 "Boolean: %b[No,Yes]\n"
                 "Integer: %i\n"
                 "Real 1: %r\n"
                 "Sep1 %t\n"
                 "Integer: %i[0,255]\n"
                 "Real 2: %r[-1.5,1.5,0.05]\n"
                 "Sep2 %t\n"
                 "Angle: %a[0,360]\n"
                 "String: %s\n"
                 "Options: %o|item0|item1|item2|\n"
                 "List: %l|item0|item1|item2|item3|item4|item5|item6|\n"
                 "File: %f[OPEN|*.bmp;*.jpg|CURRENT|NO|NO]\n"
                 "Color: %c{Color Tip}\n"
                 "Font: %n\n"
                 "Sep3 %t\n"
                 "Multiline: %m\n",
                 &pboolean, &pinteger, &preal, &pinteger2, &preal2, &pangle, pstring,
                 &poptions, &plist, file_name, pcolor, pfont, pstring2, NULL))
  return;
```



See Also

IupScanf, IupGetColor, IupMask, IupVal, IupDial, IupList, IupFileDlg.

IupGetText

Shows a modal dialog to edit a multiline text.

Creation and Show

```
int IupGetText(const char* title, char *text, int maxsize); [in C]
iup.GetText(title, text: string[, maxsize: number]) -> (text: string) [in Lua]
```

text: It contains the initial value of the text and the returned text. It must have room for the edited string with at least **maxsize** lenght. In Lua the default **maxsize** if 10240. (**maxsize** since 3.17)

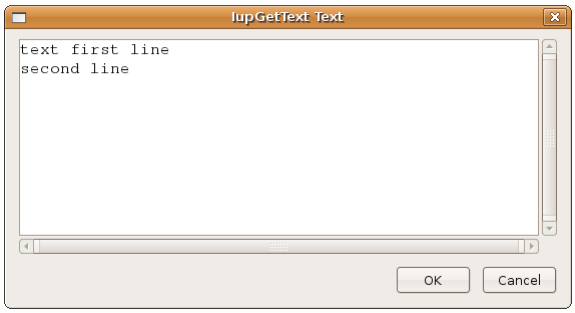
Returns: a non zero value if successful. In Lua returns the text or nil if an error occurred.

Notes

The function does not allocate memory space to store the text entered by the user. Therefore, the text parameter must be large enough to contain the user input. The returned string is limited to maxsize characters. In Lua the

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

Examples



See Also

IupMessage, IupScanf, IupListDialog, IupAlarm, IupSetLanguage.

IupListDialog

Shows a modal dialog to select items from a simple or multiple selection list.

Creation and Show

```
int IupListDialog(int type, const char *title, int size, const char** list, int op, int max_col, int max_lin, int* marks); [in C]
iup.ListDialog(type: number, title: string, size: number, list: table of strings, op: number, max_col: number, max_lin: number, marks: table of numbers) -> status: number
```

type: 1=simple selection; 2=multiple selection
title: Text for the dialog's title
size: Number of options
list: List of options. Must have **size** elements
op: Initial selected item when type=1. starts at 1 (note that this index is different from the return value, kept for compatibility reasons)
max_col: number of visible columns in the list
max_lin: number of visible lines in the list
marks: List of the items selection state, used only when type=2. Can be NULL when type=1. When type=2 must have **size** elements

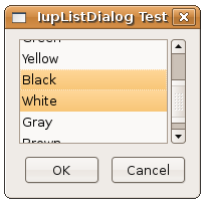
Returns: When type=1, the function returns the number of the selected option (starts at 0), or -1 if the user cancels the operation.
When type=2, the function returns -1 when the user cancels the operation. If the user does not cancel the operation the function returns 1 and the **marks** parameter will have value 1 for the options selected by the user and value 0 for non-selected options. In Lua, the input table mark is changed.

Notes

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

Examples

Browse for Example Files



See Also

IupMessage, IupScanf, IupGetFile, IupAlarm

IupMessage

Shows a modal dialog containing a message. It simply creates and popup a **IupMessageDlg**.

Creation and Show

```
void IupMessage(const char *title, const char *message); [in C]
iup.Message(title: string, message: string) [in Lua]
```

title: dialog title
message: text message contents

Notes

The **IupMessage** function shows a dialog centralized on the screen, showing the message and the "OK" button. The '\n' character can be added to the message to indicate line change.

In C there is an utility function to help build the message string, it accepts the same format as the C **sprintf**:

```
void IupMessagef(const char *title, const char *format, ...); [in C]
```

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined. It also uses a global attribute called "ICON" as the dialog icon if it is defined (used only in Motif, in Windows MessageBox does not have an icon in the title bar).

Examples

Browse for Example Files

See Also

IupGetFile, IupScanf, IupListDialog, IupAlarm, IupMessageDlg

IupScanf

Shows a modal dialog for capturing values with a format similar to the scanf function in the C stdio library.

It is recommended that new applications use the IupGetParam dialog instead.

Creation and Show

```
int IupScanf(const char *format, ...); [in C]
iup.Scanf(format: string, ...) -> (...) [in Lua]
```

format: Reading format
...: List of variables

Returns: In C the number of successfully read fields, or -1 when the user has canceled the operation. In Lua, the code is not returned, the values are returned by the function in the same order they were passed, or nil when the user has canceled the operation.

Notes

The **fmt** format must include a title and the descriptions of the variable fields to be read, using the following syntax:

- First line: Window title followed by '\n'

- Following lines: Must be specified for each variable to be read, in the following format:

"**text**%**t**.**v**%**f**\n", where:

text is a descriptive text, to be placed to the left of the text field in a label.
t is the maximum number of characters allowed
v is the number of visible characters in the text field
f is the type (char, float, etc), in the C format for I/O services (d,i,o,u,x,X,e,f,g,E,G,s, and the modifiers l,h)

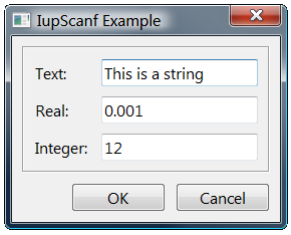
All the fields use a text box for input. If you need better control of what characters the user enters, you should use IupGetParam. This other dialog also has many other resources not available in **IupScanf**.

The dialog uses a global attribute called "PARENTDIALOG" as the parent dialog if it is defined. It also uses a global attribute called "ICON" as the dialog icon if it is defined.

Examples

Captures an integer number, a floating-point value and a character string.

Browse for Example Files



See Also

IupGetFile, IupMessage, IupListDialog, IupAlarm, IupGetParam

See Also

IupDialog, IupShow, IupShowXY, IupPopup

See Also

IupDialog, IupShow, IupShowXY, IupPopup, IupLayoutDialog

Layout Composition

Abstract Layout

Most interface toolkits employ the concrete layout model, that is, control positioning in the dialog is absolute in coordinates relative to the upper left corner of the dialog's client area. This makes it easy to position the controls on it by using an interactive tool usually provided with the system. It is also easy to dimension them. Of course, this positioning intrinsically depends on the graphics system's resolution. Moreover, when the dialog size is altered, the elements remain on the same place, thus generating an empty area below and to the right of the elements. Besides, if the graphics system's resolution changes, the dialog inevitably will look larger or smaller according to the resolution increase or decrease.

IUP implements an abstract layout concept, in which the positioning of controls is done relatively instead of absolutely. For such, composition elements are necessary for composing the interface elements. They are boxes and fillings invisible to the user, but that play an important part. When the dialog size changes, these containers expand or retract to adjust the positioning of the controls to the new situation.

Watch the codes below. The first one refers to the creation of a dialog for the Microsoft Windows environment using its own resource API. The second uses IUP. Note that, apart from providing the specification greater flexibility, the IUP specification is simpler, though a little larger. In fact, creating a dialog on IUP with several elements will force you to plan your dialog more carefully – on the other hand, this will actually make its implementation easier.

Moreover, this IUP dialog has an indirect advantage: if the user changes its size, the elements (due to being positioned on an abstract layout) are automatically re-positioned horizontally.

The composition elements includes vertical boxes (**vbox**), horizontal boxes (**hbox**) and filling (**fill**). There is also a depth box (**zbox**) in which layers of elements can be created for the same dialog, and the elements in each layer are only visible when that given layer is active.

in Windows	in IupLua
	dialog = iup.dialog

```
dialog DIALOG 0, 0, 117, 32
STYLE WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
      WS_CAPTION | WS_SYSMENU |
      WS_THICKFRAME
CAPTION "Title"
BEGIN
    PUSHBUTTON "Ok",IDOK,17,9,33,15
    PUSHBUTTON "Cancel",IDCANCEL,66,9,33,15
END
```

```
{
  iup.hbox
  {
    iup.fill{},
    iup.button{title="Ok",size="40"},
    iup.button{title="Cancel",size="40"},
    iup.fill{}
    ;margin="15x15", gap="10"
  }
  ;title="Title"
}
```

Now see the same dialog in LED and in C:

in LED

```
dialog = DIALOG[TITLE="Title"]
(
  HBOX[MARGIN="15x15", GAP="10"]
  (
    FILL(),
    BUTTON[SIZE="40"]("Ok",do_nothing),
    BUTTON[SIZE="40"]("Cancel",do_nothing),
    FILL()
  )
)
```

in C

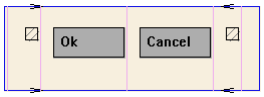
```
dialog = IupSetAttributes(IupDialog
(
  IupSetAttributes(IupHbox
  (
    IupFill(),
    IupSetAttributes(IupButton("Ok", NULL), "SIZE=40"),
    IupSetAttributes(IupButton("Cancel", NULL), "SIZE=40"),
    IupFill(),
    NULL
  ), "MARGIN=15x15, GAP=10")),
), "TITLE=Title")
```

Following, the abstract layout representation of this dialog:

Layout Hierarchy

```
Dialog
  HBox
    Fill
    Button
    Button
    Fill
```

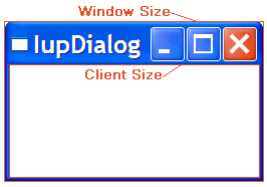
Layout Visualization

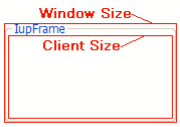


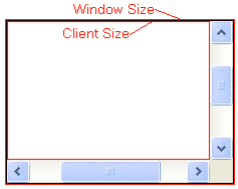
Layout Guide

Native Sizes (Window and Client)

Because of the dynamic nature of the abstract layout IUP elements have implicit many types of size. But the native elements have only two types of size: **Window** and **Client**. The **Window** size reflects the bounding rectangle of the element. The **Client** size reflects the inner size of the window that excludes the decorations and margins. For many elements these two sizes are equal, but for many containers they are quite different. See some examples below.







The IUP sizes (**User**, **Natural** and **Current**) described below are all related to the **Window** size.

The native **Client** size is used only internally to reposition the elements in the abstract layout, but it is available using the CLIENTSIZE attribute.

IUP Sizes

Natural Size

IUP does not require that the application specifies the size of any element. The sizes are automatically calculated so the contents of each element is fully displayed. This size is called **Natural** size. The **Natural** size is calculated just before the element is mapped to the native system and every time **IupMap** is called, even if the element is already mapped.

The **Natural** size of a container is the size that allows all the elements inside the container to be fully displayed. Then the **Natural** size is calculated from the inner element to the outer element (the dialog). Important: even if the element is invisible its size will be included in the size of its containers, except when FLOATING=Yes.

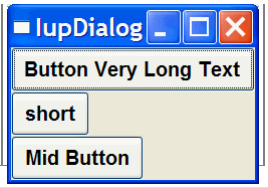
So consider the following code and its result. Each button size is large enough to display their respective text. If the dialog size is increased or reduced by the size handlers in the dialog borders the buttons do not move or change their sizes.

But notice that some controls do not have contents that can provide a **Natural** size. In this case they usually have SIZE or RASTERSIZE pre-set.

To obtain the last computed **Natural** size of the control in pixels, use the read-only attribute NATURALSIZE (since 3.6).

```
dlg = iup.dialog
{
  iup.vbox
  {
    iup.button{title="Button Very Long Text"},
    iup.button{title="short"},
```

```
    iup.button{title="Mid Button"}
  }
  ;title="IupDialog", font="Helvetica, Bold 14"
}
dlg:show()
```



User Size

When the application defines the SIZE or RASTERSIZE attributes, it changes the **User** size in IUP. The initial internal value is "0x0". When set to NULL the **User** size is internally set to "0x0". If the element is not mapped then the returned value by SIZE or RASTERSIZE is the **User** size, if the element is mapped then the returned value is the **Current** size. To obtain the **User** size after the element is mapped use the USERSIZE attribute (since 3.12).

By default the layout computation uses the **Natural** size of the element to compose the layout of the dialog, but if the **User** size is defined then it is used instead of the **Natural** size. In this case the **Natural** size is not even computed. But there are two exceptions.

If the element is a container (not including the dialog) the **User** size will be used instead of the **Natural** size only if bigger than the **Natural** size. So for containers the **User** size will also act as a minimum value for **Natural** size.

For the dialog, if the **User** size is defined then it is used instead of the **Natural** size, but the **Natural** size of the dialog is always computed. And if the **User** size is not defined, the **Natural** size is used only if bigger than the **Current** size, so in this case the dialog will always increase its size to fit all its contents. In other words, in this case the dialog will not shrink its Current size unless the **User** size is defined. See the SHRINK attribute guide bellow for an alternative.

When the user is interactively changing the dialog size the **Current** size is updated. But the dialog contents will always occupy the Natural size available, being smaller or bigger than the dialog **Current** size.

When SIZE or RASTERSIZE attributes are set for the dialog (changing the **User** size) the **Current** size is also reset to "0x0". Allowing the application to force an update of its **Window** size. To only change the **User** size in pixels, without resetting the **Current** size, set the USERSIZE attribute (since 3.12).

Current Size

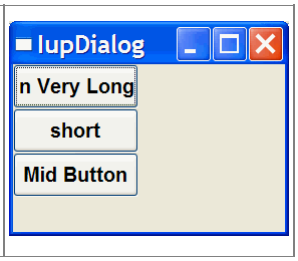
After the **Natural** size is calculated for all the elements in the dialog, the the **Current** size is set based on the available space in the dialog. So the **Current** size is set from the outer element (the dialog) to the inner element, in opposite of what it is done for the **Natural** size.

After all the elements have their **Current** size updated, the elements positions are calculated, and finally, after the element is mapped, the **Window** size and position are set for the native elements. The **Window** size is set exactly to the **Current** size.

After the element is mapped the returned value for SIZE or RASTERSIZE is the **Current** size. It actually returns the native **Window** size of the element. Before mapping, the returned value is the **User** size.

Defining the SIZE attribute of the buttons in the example we can make all have the same size. (In the following example the dialog size was changed after it was displayed on screen)

```
dlg = iup.dialog
{
  iup.vbox
  {
    iup.button{title="Button Very Long Text", size="50x"},
    iup.button{title="short", size="50x"},
    iup.button{title="Mid Button", size="50x"}
  }
  ;title="IupDialog", font="Helvetica, Bold 14"
}
dlg:show()
```



So when EXPAND=NO (see below) for elements that are not containers if **User** size is defined then the **Natural** size is ignored.

If you want to adjust sizes in the dialog do it after the layout size and positioning are done, i.e. after the dialog is mapped or after **IupRefresh** is called.

EXPAND

Another way to increase the size of elements is to use the EXPAND attribute. When there is room in the container to expand an element, the container layout will expand the elements that have the EXPAND attribute set to YES, HORIZONTAL or VERTICAL accordingly, even if they have the **User** size defined.

The default is EXPAND=NO, but for containers is EXPAND=YES.

Using EXPAND in the example, we obtain the following result:

```
dlg = iup.dialog
{
  iup.vbox
  {
    iup.button{title="Button Very Long Text"},
    iup.button{title="short", expand="HORIZONTAL"},
    iup.button{title="Mid Button", expand="HORIZONTAL"}
  }
  ;title="IupDialog", font="Helvetica, Bold 14"
}
dlg:show()
```

So for elements that are NOT containers, when EXPAND is enabled the **Natural** size and the **User** size are ignored.

For containers the default behavior is to always expand or if expand is disabled they are limited to the **Natural** size. As a consequence (if the **User** size is not defined in all the elements) the dialog contents can only expand and its minimum size is the **Natural** size, even if EXPAND is enabled for its elements. In fact the actual dialog size can be smaller, but its contents will stop to follow the resize and they will be clipped at right and bottom.

If the expansion is in the same direction of the box, for instance expand="VERTICAL" in the Vbox of the previous example, then the expandable elements will receive equal spaces to expand according to the remaining empty space in the box. This is why elements in different boxes does not align perfectly when EXPAND is set.

SHRINK

To reduce the size of the dialog and its containers to a size smaller than the **Natural** size the SHRINK attribute of the dialog can be used. If set to YES all the containers of the dialog will be able to reduce its size. But be aware that elements may overlap and the layout result could be visually bad if the dialog size is smaller than its **Natural** size.

Notice that in the example the dialog initial size will be 0x0 because it is not defined. The picture shown was captured after manually resizing the dialog. So when using SHRINK usually you will also need to set the dialog initial size.

```
dlg = iup.dialog
{
  iup.vbox
  {
    iup.button{title="Button Very Long Text"},
    iup.button{title="short", expand="HORIZONTAL"},
    iup.button{title="Mid Button", expand="HORIZONTAL"}
  }
  ;title="IupDialog", shrink="yes", font="Helvetica, Bold 14"
}
dlg:show()
```

Layout Hierarchy

The layout of the elements of a dialog in IUP has a natural hierarchy because of the way they are composed together.

To create a node simply call one of the pre-defined constructors like **IupLabel**, **IupButton**, **IupCanvas**, and so on. To create a branch just call the constructors of containers like **IupDialog**, **IupFrame**, **IupVBox**, and so on. Internally they all call IupCreate to create branches or nodes. To destroy a node or branch call IupDestroy.

Some of the constructors already append children to its branch, but you can add other children using IupAppend or IupInsert. To remove from the tree call IupDetach.

For the element to be visible IupMap must be called so it can be associated with a native control. **IupShow**, **IupShowXY** or **IupPopup** will automatically call **IupMap** before showing a dialog. To remove this association call IupUnmap.

But there is a call order to be able to call theses functions that depends on the state of the element. As you can see from these functions there are 3 states: **created**, **appended** and **mapped**. From **created** to **mapped** it is performed one step at a time. Even when the constructor receives the children as a parameter **IupAppend** is called internally. When you **detach** an element it will be automatically **unmapped** if necessary. When you **destroy** an element it will be automatically **detached** if necessary. So explicitly or implicitly, there will be always a call to:

```
IupCreate  -> IupAppend -> IupMap
IupDestroy <- IupDetach <- IupUnmap
```

A more simple and fast way to move an element from one position in the hierarchy to another is using IupReparent.

The dialog is the root of the hierarchy tree. To retrieve the dialog of any element you can simply call IupGetDialog, but there are other ways to navigate in the hierarchy tree.

To get all the children of a container call IupGetChild or IupGetNextChild. To get just the next control with the same parent use IupGetBrother. To get the parent of a control call IupGetParent.

In Lua, if the element was created in Lua, you can access any child of the element using the notation "elem[n][n]....", where n is the index of the child. For example:

```
dlg = iup.dialog
{
  iup.hbox
  {
    iup.button{title="Ok"},
    iup.button{title="Cancel"},
  }
}
cancel_button = dlg[1][2]
```

Layout Display

The layout size and positioning is automatically updated by **IupMap**. **IupMap** also updates the dialog layout even if it is already mapped, so using it or using **IupShow**, **IupShowXY** or **IupPopup** (they all call **IupMap**) will also update the dialog layout. The layout size and positioning can be manually updated using IupRefresh, even if the dialog is not mapped.

After changing containers attributes or element sizes that affect the layout the elements are NOT immediately repositioned. Call **IupRefresh** for an element inside the dialog to update the dialog layout.

The Layout update is done in two phases. First the layout is computed, this can be done without the dialog being mapped. Second is the native elements update from the computed values.

The Layout computation is done in 3 steps: **Natural** size computation, update the **Current** size and update the position.

- The **Natural** size computation is done from the inner elements up to the dialog (first for the children then the element). **User** size (set by RASTERSIZE or SIZE) is used as the **Natural** size if defined, if not usually the contents of the element are used to calculate the **Natural** size.
- Then the **Current** size is computed starting at the dialog down to the inner elements on the layout hierarchy (first the element then the children). Children **Current** size is computed according to layout distribution and containers decoration. At the children if EXPAND is set, then the size specified by the parent is used, else the natural size is used.
- Finally the position is computed starting at the dialog down to the inner elements on the layout hierarchy, after all sizes are computed.

Element Update

Usually IUP automatically updates everything for the application, for instance there is no need to force a display update after an attribute is changed. But there are some situations where you need to force an update. Here is a summary of the functions that can be used to update an element state:

IupUpdate - update the element look by letting the system to schedule a redraw. Rarely used.

IupRedraw - has the same effect of **IupUpdate** but forces the element to redraw now.

IupRefresh - if the application changed some attribute that affects the natural size, for instance SIZE or RASTERSIZE among others, the actual element size is NOT immediately updated. That's because it can affect the size and position of other elements in the dialog. **IupRefresh** will force an update in the layout of the hole dialog, and of course if an element has its size changed its appearance will be automatically updated.

IupFlush - process all events that are waiting to be processed. When you set an attribute, a system event is generated, but it will wait until is processed by the event loop. Sometimes the application needs an immediate result, so calling **IupFlush** will process that event but it will also process every other event that was waiting to be processed, so other callbacks could be trigger during **IupFlush** call.

IupAppend

Inserts an interface element at the end of the container, **after** the last element of the container. Valid for any element that contains other elements like dialog, frame, hbox, vbox, zbox or menu.

Parameters/Return

```
Ihandle* IupAppend(Ihandle* ih, Ihandle* new_child); [in C]
iup.Append(ih, new_child: ihandle) -> (parent: ihandle) [in Lua]
```

```
or ih:append(new_child: ihandle) -> (parent: ihandle) [in Lua]
```

ih: Identifier of a container like hbox, vbox, zbox and menu.
new_child: Identifier of the element to be inserted.

Returns: the actual **parent** if the interface element was successfully inserted. Otherwise returns NULL (nil in Lua). Notice that the desired parent can contains a set of elements and containers where the child will be actually attached so the function returns the actual parent of the element.

Notes

This function can be used when elements that will compose a container are not known *a priori* and should be dynamically constructed.

The new child can NOT be mapped. It will NOT map the new child into the native system. If the parent is already mapped you must explicitly call **IupMap** for the appended child.

If the actual parent is a layout box (**IupVbox**, **IupHbox** or **IupZbox**) and you try to append a child that it is already at the parent child list, then the child is moved to the last child position.

The elements are NOT immediately repositioned. Call **IupRefresh** for the container (or any other element in the dialog) to update the dialog layout.

See Also

IupDetach, IupInsert, IupHbox, IupVbox, IupZbox, IupMenu, IupMap, IupUnmap, IupRefresh

IupDetach

Detaches an interface element from its parent.

Parameters/Return

```
void IupDetach(Ihandle *child); [in C]
iup.Detach(child: ihandle) [in Lua]
or child:detach() [in Lua]
```

child: Identifier of the interface element to be detached.

Notes

It will automatically call **IupUnmap** to **unmap** the element if necessary, and then **detach** the element.

If left **detached** it is still necessary to call **IupDestroy** to **destroy** the IUP element.

The elements are NOT immediately repositioned. Call **IupRefresh** for the container (or any other element in the dialog) to update the dialog layout.

When the element is mapped some attributes are stored only in the native system. If the element is **unmaped** those attributes are lost. Use the function IupSaveClassAttributes when you want to **unmap** the element and keep its attributes.

See Also

IupAppend, IupInsert, IupRefresh, IupUnmap, IupCreate, IupDestroy

IupInsert (Since 3.0)

Inserts an interface element **before** another child of the container. Valid for any element that contains other elements like dialog, frame, hbox, vbox, zbox, menu, etc.

Parameters/Return

```
Ihandle* IupInsert(Ihandle* ih, Ihandle* ref_child, Ihandle* new_child); [in C]
iup.Insert(ih, ref_child, new_child: ihandle) -> (parent: ihandle) [in Lua]
or ih:insert(ref_child, new_child: ihandle) -> (parent: ihandle) [in Lua]
```

ih: Identifier of a container like hbox, vbox, zbox and menu.
ref_child: Identifier of the element to be used as reference. Can be NULL to insert as the first element.
new_child: Identifier of the element to be inserted before the reference.

Returns: the actual **parent** if the interface element was successfully inserted. Otherwise returns NULL (nil in Lua). Notice that the desired parent can contains a set of elements and containers where the child will be actually attached so the function returns the actual parent of the element.

Notes

This function can be used when elements that will compose a container are not known *a priori* and should be dynamically constructed.

The new child can NOT be mapped. It will NOT map the new child into the native system. If the parent is already mapped you must explicitly call **IupMap** for the appended child.

If the actual parent is a layout box (**IupVbox**, **IupHbox** or **IupZbox**) and you try to insert a child that it is already at the parent child list, then the child is moved to the insert position.

The elements are NOT immediately repositioned. Call **IupRefresh** for the container* to update the dialog layout (* or any other element in the dialog).

See Also

IupAppend, IupDetach, IupHbox, IupVbox, IupZbox, IupMenu, IupMap, IupUnmap, IupRefresh

IupReparent (Since 3.0)

Moves an interface element from one position in the hierarchy tree to another.

Both **new_parent** and **child** must be mapped or unmapped at the same time.

If **ref_child** is NULL, then it will *append* the **child** to the **new_parent**. If **ref_child** is NOT NULL then it will *insert* **child** before **ref_child** inside the **new_parent**.

Parameters/Return

```
int IupReparent(Ihandle* child, Ihandle* new_parent, Ihandle* ref_child); [in C]
iup.Reparent(child, new_parent, ref_child: ihandle) -> error: number [in Lua]
```

child: Identifier of the element to be moved.
new_parent: Identifier of the new parent.
ref_child: Identifier of the element to be used as reference, where the child will be inserted before it. Can be NULL. (since 3.3)

Returns: IUP_NOERROR if successfully, IUP_ERROR if failed.

Notes

This function is faster and easier than doing the sequence **unmap**, **detach**, **append/insert** and **map**.

The elements are NOT immediately repositioned. Call **IupRefresh** for the container (or any other element in the dialog) to update the dialog layout.

Motif does not support the re-parent function, but we simulate a re-parent doing a **unmap**/**map** sequence. But some attributes may be lost during the operation, mostly attributes that are id dependent.

See Also

IupAppend, IupInsert, IupDetach, IupMap, IupUnmap, IupRefresh

IupGetParent

Returns the parent of a control.

Parameters/Return

```
Ihandle* IupGetParent(Ihandle *ih); [in C]
iup.GetParent(ih: ihandle) -> parent: ihandle [in Lua]
```

ih: identifier of the interface element.

Returns: the handle of the parent or NULL if does not have a parent.

See Also

IupGetChild, IupGetNextChild, IupGetBrother

IupGetChild

Returns the a child of the control given its position.

Parameters/Return

```
Ihandle *IupGetChild(Ihandle* ih, int pos); [in C]
iup.GetChild(ih: ihandle, pos: number) -> child: ihandle [in Lua]
```

ih: identifier of the interface element.
pos: position of the desire child starting at 0.

Returns: the child or NULL if there is none.

Notes

This function will return the children of the control in the exact same order in which they were assigned.

See Also

IupGetChildPos, IupGetNextChild, IupGetBrother, IupGetParent

IupGetChildPos (since 3.0)

Returns the position of a child of the given control.

Parameters/Return

```
int IupGetChildPos(Ihandle* ih, Ihandle* child); [in C]
iup.GetChildPos(ih, child: ihandle) ->  pos: number [in Lua]
```

ih: identifier of the interface element.

Returns: the position of the desire child starting at 0, or -1 if child not found.

Notes

This function will return the children of the control in the exact same order in which they were assigned.

See Also

IupGetChild, IupGetChildCount, IupGetNextChild, IupGetBrother, IupGetParent

IupGetChildCount(since 3.0)

Returns the number of children of the given control.

Parameters/Return

```
int IupGetChildCount(Ihandle* ih); [in C]
iup.GetChildCount(ih: ihandle) ->  pos: number [in Lua]
```

ih: identifier of the interface element.

Returns: the number of children.

See Also

IupGetChildPos, IupGetChild, IupGetNextChild, IupGetBrother, IupGetParent

IupGetNextChild

Returns the a child of the given control given its brother.

Parameters/Return

```
Ihandle *IupGetNextChild(Ihandle* ih, Ihandle* child); [in C]
iup.GetNextChild(ih, child: ihandle) -> next_child: ihandle [in Lua]
```

ih: identifier of the interface element. Can be NULL if child not NULL.
child: Identifier of the child brother to be used as reference. To get the first child use NULL.

Returns: the handle of the child or NULL.

Notes

This function will return the children of the control in the exact same order in which they were assigned. If child in not NULL then it returns exactly the same result as IupGetBrother.

Example

```
/* Lists all children of a IupVbox */
```

```
#include <stdio.h>
#include "iup.h"

int main(int argc, char* argv[])
{
  Ihandle *dialog, *bt, *lb, *vbox, *child;

  IupOpen(&argc, &argv);

  bt = IupButton("Button", NULL);
  lb = IupLabel("Label");

  vbox = IupVbox(bt, lb, NULL);

  dialog = IupDialog(vbox);
  IupShow(dialog);

  child = IupGetNextChild(vbox, NULL);

  while(child)
  {
    printf("vbox has a child of type %s\n", IupGetClassName(child));
    child = IupGetNextChild(NULL, child);
  }

  IupMainLoop();
  IupClose();

  return 0;
}
```

See Also

IupGetBrother, IupGetParent, IupGetChild

IupGetBrother

Returns the brother of an element.

Parameters/Return

```
Ihandle* IupGetBrother(Ihandle* ih); [in C]
iup.GetBrother(ih: ihandle) -> brother: ihandle [in Lua]
```

ih: identifier of the interface element.

Returns: the brother or NULL if there is none.

See Also

IupGetChild, IupGetNextChild, IupGetParent

IupGetDialog

Returns the handle of the dialog that contains that interface element. Works also for children of a menu that is associated with a dialog.

Parameters/Return

```
Ihandle* IupGetDialog(Ihandle *ih); [in C]
iup.GetDialog(ih: ihandle) -> (ih: ihandle) [in Lua]
```

ih: Identifier of an interface element.

Returns: the handle of the dialog or NULL if not found.

IupGetDialogChild (since 3.0)

Returns the identifier of the child element that has the NAME attribute equals to the given value on the same dialog hierarchy. Works also for children of a menu that is associated with a dialog.

Parameters/Return

```
Ihandle* IupGetDialogChild(Ihandle *ih, const char* name); [in C]
iup.GetDialogChild(ih: ihandle, name: string) -> (ih: ihandle) [in Lua]
```

ih: Identifier of an interface element that belongs to the hierarchy.
name: name of the control to be found

Returns: NULL if not found.

Notes

This function will only found the child if the NAME attribute is set at the control.

Before the dialog is mapped the function searches the hierarchy, even if the hierarchy does not belongs to a dialog yet, but after the child is mapped the result is immediate (the hierarchy is not searched).

See Also

NAME

IupRefresh

Updates the size and layout of all controls in the same dialog.

To be used after changing size attributes, or attributes that affect the size of the control. Can be used for any element inside a dialog, but the layout of the dialog and all controls will be updated. It can change the layout of all the controls inside the dialog because of the dynamic layout positioning.

Parameters/Return

```
void IupRefresh(Ihandle *ih); [in C]
iup.Refresh(ih: ihandle) [in Lua]
```

ih: identifier of the interface element.

Notes

Can be used for any control, but it will always affect the whole dialog. Can be called even if the dialog is not mapped.

To refresh the layout of only a subset of the dialog use IupRefreshChildren.

After the layout is computed, the position and size attributes are all updated. If the elements are mapped then they are immediately repositioned, if the dialog is visible then the change will be immediately reflected on the display.

This function will NOT change the size of the dialog, **except** if the SIZE or RASTERSIZE attributes of the dialog where changed before the call. For instance, if you also want to change the size of the dialog then you can do:

```
IupSetAttribute(dialog, "SIZE", ...);
IupRefresh(dialog);
```

So the dialog will be resized for the new **User** size, if the new size is NULL the dialog will be resized to the **Natural** size that include all the elements.

Changing the size of elements without changing the dialog size may position some controls outside the dialog area at the left or bottom borders (the elements will be cropped at the dialog borders by the native system).

IupMap also updates the dialog layout, but only when called for the dialog itself, even if the dialog is already mapped. Since **IupShow**, **IupShowXY** and **IupPopup** call **IupMap**, then they all will always update the dialog layout before showing it, even also if the dialog is already visible.

See Also

SIZE, IupMap, IupRefreshChildren

IupRefreshChildren (Since 3.3)

Updates the size and layout of controls after changing size attributes, or attributes that affect the size of the control. Can be used for any element inside a dialog, only its children will be updated. It can change the layout of all the controls inside the given element because of the dynamic layout positioning.

Parameters/Return

```
void IupRefreshChildren(Ihandle *ih); [in C]
iup.RefreshChildren(ih: ihandle) [in Lua]
```

ih: identifier of the interface element.

Notes

The given element must be a container. It must be inside a dialog hierarchy and must be mapped. It can not be a dialog. For dialogs use **IupRefresh**.

The children are immediately repositioned, if the dialog is visible then the change will be immediately reflected on the display.

This function will NOT change the size of the given element, even if the natural size of its children would increase its natural size.

If your dialog has too many controls and you want to hide or destroy some, then add some other in the same place, so you actually know that the dialog layout would not change, this is a much faster function than **IupRefresh**.

See Also

IupRefresh

Controls

IUP contains several user interface controls. The library's main characteristic is the use of native elements. This means that the drawing and management of a button or text box is done by the native interface system, not by IUP. This makes the application's appearance more similar to other applications in that system. On the other hand, the application's appearance can vary from one system to another.

But this is valid only for the standard controls, many additional controls are drawn by IUP. Composition controls are not visible, so they are independent from the native system.

Each control has an unique creation function, and all of its management is done by means of **attributes** and **callbacks**, using functions common to all the controls. This simple but powerful approach is one of the advantages of using IUP.

Controls are automatically destroyed when the dialog is destroyed.

child, ... : List of the identifiers that will be placed in the box. NULL must be used to define the end of the list in C. It can be empty in C or Lua, not in LED.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

EXPAND (non inheritable): The default value is "YES".

SIZE / RASTERSIZE (non inheritable): Must be defined for each child. If not defined for the box, then it will be the bounding box that includes all children in their position.

WID (read-only): returns -1 if mapped.

FONT, CLIENTSIZE, CLIENTOFFSET, POSITION, MINSIZE, MAXSIZE: also accepted.

Attributes (at Children)

CX, CY (non inheritable) **(at children only)**: Position in pixels of the child relative to the top-left corner of the box. Must be set for each child inside the box.

Notes

The box can be created with no elements and be dynamic filled using IupAppend or IupInsert.

Examples

Browse for Example Files

See Also

IupVbox, IupHbox

IupFill

Creates void element, which dynamically occupies empty spaces always trying to expand itself. Its parent should be an **IupHbox,** an **IupVbox** or a **IupGridBox**, or else this type of expansion will not work. If an EXPAND is set on at least one of the other children of the box, then the fill expansion is ignored.

It does not have a native representation.

Creation

```
Ihandle* IupFill(void); [in C]
iup.fill{} -> ih: ihandle [in Lua]
fill() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

EXPAND (non inheritable)(read-only): If **User** size is not defined, then when inside a **IupHbox/IupGridBox** EXPAND is HORIZONTAL, when inside a **IupVbox** EXPAND is VERTICAL. If **User** size is defined then EXPAND is NO.

SIZE / RASTERSIZE (non inheritable): Defines the width, if inside a **IupHbox**, or the height, if it is inside a **IupVbox**. The standard format "wxh" can also be used, but width will be ignored if inside a **IupVbox** and height will be ignored if inside a **IupHbox** (since 3.3). When consulted behaves as the standard SIZE/RASTERSIZE attributes.

WID (read-only): returns -1 if mapped.

FONT, POSITION, MINSIZE, MAXSIZE: also accepted.

Examples

Browse for Example Files

See Also

IupHbox, IupVbox.

See Also

IupVbox, IupHbox

See Also

IupZbox, IupVBox

See Also

IupZbox, IupHbox

IupZbox

Creates a void container for composing elements in hidden layers with only one layer visible. It is a box that piles up the children it contains, only the one child is visible.

It does not have a native representation.

Zbox works by changing the VISIBLE attribute of its children, so if any of the grand children has its VISIBLE attribute directly defined then Zbox will NOT change its state.

Creation

```
Ihandle* IupZbox (Ihandle *child, ...); [in C]
Ihandle* IupZboxv (Ihandle **children); [in C]
iup.zbox{child, ... : ihandle} -> (ih: ihandle) [in Lua]
zbox(child, ...) [in LED]
```

child, ... : List of the elements that will be placed in the box. NULL must be used to define the end of the list in C. It can be empty in C or Lua, not in LED.

IMPORTANT: in C, each element must have a name defined by IupSetHandle. In Lua a name is always automatically created, but you can change it later.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

ALIGNMENT (non inheritable): Defines the alignment of the visible child. Possible values:

"NORTH", "SOUTH", "WEST", "EAST",
"NE", "SE", "NW", "SW",
"ACENTER".

Default: "NW".

EXPAND (non inheritable): The default value is "YES".

VALUE (non inheritable): The visible child accessed by its name. The value passed must be the name of one of the children contained in the zbox. Use IupSetHandle or IupSetAttributeHandle to associate a child to a name. In Lua you can also use the element reference directly. When the value is changed the selected child is made visible and all other children are made invisible, regardless their previous visible state.

VALUE_HANDLE (non inheritable): The visible child accessed by its handle. The value passed must be the handle of a child contained in the zbox. When the zbox is created, the first element inserted is set as the visible child. (since 3.0)

VALUEPOS (non inheritable): The visible child accessed by its position. The value passed must be the index of a child contained in the zbox, starting at 0. When the zbox is created, the first element inserted is set as the visible child. (since 3.0)

SIZE / RASTERSIZE (non inheritable): The default size is the smallest size that fits its largest child. All child elements are considered even invisible ones, except when FLOATING=YES in a child.

WID (read-only): returns -1 if mapped.

FONT, CLIENTSIZE, CLIENTOFFSET, POSITION, MINSIZE, MAXSIZE: also accepted.

Attributes (at Children)

FLOATING (non inheritable) **(at children only)**: If a child has FLOATING=YES then its size and position will be ignored by the layout processing. Default: "NO". (since 3.0)

Notes

The box can be created with no elements and be dynamic filled using IupAppend or IupInsert.

Its children automatically receives a name when the child is appended or inserted into the tabs. (since 3.16)

The ZBOX relies on the VISIBLE attribute. If a child that is hidden by the zbox has its VISIBLE attribute changed then it can be made visible regardless of the zbox configuration. For the zbox behave as a **IupTabs** use native containers as immediate children of the zbox, like **IupScrollBox**, **IupTabs**, **IupFrame** or **IupBackgroundBox**.

Examples

Browse for Example Files

See Also

IupHbox, IupVBox

IupRadio

Creates a void container for grouping mutual exclusive toggles. Only one of its descendent toggles will be active at a time. The toggles can be at any composition.

It does not have a native representation.

Creation

```
Ihandle* IupRadio(Ihandle *child); [in C]
iup.radio{child: ihandle} -> (ih: ihandle) [in Lua]
radio(child) [in LED]
```

child: Identifier of an interface element. Usually it is a vbox or an hbox containing the toggles associated to the radio. It can be NULL (nil in Lua), not optional in LED.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

EXPAND (non inheritable): The default value is "YES".

VALUE (non inheritable): name identifier of the active toggle. The name is set by means of IupSetHandle. In Lua you can also use the element reference directly. When consulted if the toggles are not mapped into the native system the return value may be NULL or invalid.

VALUE_HANDLE (non inheritable): Changes the active toggle. The value passed must be the handle of a child contained in the radio. When consulted if the toggles are not mapped into the native system the return value may be NULL or invalid. (since 3.0)

WID (read-only): returns -1 if mapped.

FONT, CLIENTSIZE, CLIENTOFFSET, POSITION, MINSIZE, MAXSIZE, VISIBLE: also accepted.

Notes

The radio can be created with no elements and be dynamic filled using IupAppend or IupInsert.

A toggle that is a child of an **IupRadio** automatically receives a name when its is mapped into the native system. (since 3.16)

Currently **IupFlatButton** with TOGGLE=YES, **IupToggle**, and **IupGLToggle** are affected when inside a **IupRadio**.

Examples

Browse for Example Files

IupNormalizer (since 3.0)

Creates a void container that does not affect the dialog layout. It acts by normalizing all the controls in a list so their natural size becomes the biggest natural size amongst them. All natural widths will be set to the biggest width, and all natural heights will be set to the biggest height. The controls of the list must be inside a valid container in the dialog.

Creation

```
Ihandle* IupNormalizer(Ihandle *ih_first, ...); [in C]
Ihandle* IupNormalizerv(Ihandle **ih_list); [in C]
iup.normalizer{ih_first, ...: ihandle} -> (ih: ihandle) [in Lua]
normalizer(ih_first, ...) [in LED]
```

ih_first, ... : List of the identifiers that will be normalized. NULL must be used to define the end of the list in C. It can be empty in C or Lua, not in LED.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

NORMALIZE (non inheritable): normalization direction. Can be HORIZONTAL, VERTICAL or BOTH. These are the same values of the NORMALIZESIZE attribute. Default: HORIZONTAL.

ADDCONTROL (non inheritable, write-only): Adds a control to the normalizer. The value passed must be the name of an element. Use IupSetHandle or IupSetAttributeHandle to associate an element to a name. In Lua you can also use the element reference directly.

ADDCONTROL_HANDLE (non inheritable, write-only): Adds a control to the normalizer. The value passed must be a handle of an element.

DELCONTROL (non inheritable, write-only): Removes a control from the normalizer. The value passed must be the name of an element. Use IupSetHandle or IupSetAttributeHandle to associate an element to a name. In Lua you can also use the element reference directly. (since 3.17)

DELCONTROL_HANDLE (non inheritable, write-only): Removes a control from the normalizer. The value passed must be a handle of an element. (since 3.17)

Attributes (at Controls)

NORMALIZERGROUP (non inheritable) **(at controls only)**: name of a normalizer element to which to automatically add the control. If an element with that name does not exists then one is created. In Lua you can also use the element reference directly.

Notes

It is NOT necessary to add the normalizer to a dialog hierarchy. Every time the NORMALIZE attribute is set, a normalization occurs. If the normalizer is added to a dialog hierarchy, then whenever the **Natural** size is calculated a normalization occurs, so you should add it to the hierarchy before the elements you want to normalize or its normalization will be not used.

The elements do NOT need to be children of the same parent, do NOT need to be mapped, and do NOT need to be in a complete hierarchy of a dialog.

The elements are NOT children of the normalizer, so **IupAppend**, **IupInsert** and **IupDetach** can not be used. To add or remove elements use the **ADDCONTROL** and **DELCONTROL** attributes.

Notice that the NORMALIZERGROUP attribute can simplify a lot of the process of creating a normalizer, so you do not need to list several elements from different parts of the dialog.

Has the same effect as the NORMALIZESIZE attribute of the **IupVbox** and **IupHbox** controls, but it can be used for elements with different parents, it changes the **User** size of the elements.

Examples

Here **IupNormalizer** is used to normalize the horizontal size of several labels that are in different containers. Since it needs to be done once only the **IupNormalizer** is destroyed just after it is initialized.

```
IupDestroy(IupSetAttributes(IupNormalizer(IupGetChild(hsi_vb, 0),  /* Hue Label */
                                          IupGetChild(hsi_vb, 1),  /* Saturation Label */
                                          IupGetChild(hsi_vb, 2),  /* Intensity Label */
                                          IupGetChild(clr_vb, 0),  /* Opacity Label */
                                          IupGetChild(clr_vb, 1),  /* Hexa Label */
                                          NULL), "NORMALIZE=HORIZONTAL"));
```

The following case use the internal normalizer in an Hbox:

```
button_box = IupHbox(
    IupFill(),
    button_ok,
    button_cancel,
    button_help,
    NULL);
IupSetAttribute(button_box, "NORMALIZESIZE", "HORIZONTAL");
```

See Also

IupHbox, IupVbox, IupGridBox

ih: identifier of the element that activated the event.
pos: the tab position

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

The Tabs can be created with no children and be dynamic filled using IupAppend.

The ENTERWINDOW_CB and LEAVEWINDOW_CB callbacks are called only when the mouse enter or leaves the tabs buttons area.

Its children automatically receives a name when the child is appended or inserted into the tabs.

Differently from **IupZbox**, **IupTabs** does NOT depends on the VISIBLE attribute.

In GTK, when the tabs buttons are scrolled, the active tab is also changed.

When you change the active tab the focus is usually not changed. If you want to control the focus behavior call **IupSetFocus** in the TABCHANGE_CB callback. Unfortunately this does not works in GTK and in Motif, because in both systems the focus will be set by the system after the callback is called.

Notice that there is no attribute to disable a single tab. This is a design decision of all native toolkits, not a IUP decision. It is so because a disabled tab is a confusing interface situation.

In Windows, when an **IupVal** is inside an **IupTabs**, the tabs disappear when the mouse moves over it after being used in the valuator. A workaround is to put the valuator inside an **IupFrame** and then inside the **IupTabs**, so the problem does not occur.

Utility Functions

These functions can be used to set and get attributes from the element:

```
void  IupSetAttributeId(Ihandle *ih, const char* name, int id, const char* value);
char* IupGetAttributeId(Ihandle *ih, const char* name, int id);
int   IupGetIntId(Ihandle *ih, const char* name, int id);
float IupGetFloatId(Ihandle *ih, const char* name, int id);
void  IupSetfAttributeId(Ihandle *ih, const char* name, int id, const char* format, ...);
void  IupSetIntId(Ihandle* ih, const char* name, int id, int value);
void  IupSetFloatId(Ihandle* ih, const char* name, int id, float value);
```

They work just like the respective traditional set and get functions. But the attribute string is complemented with the id value. For ex:

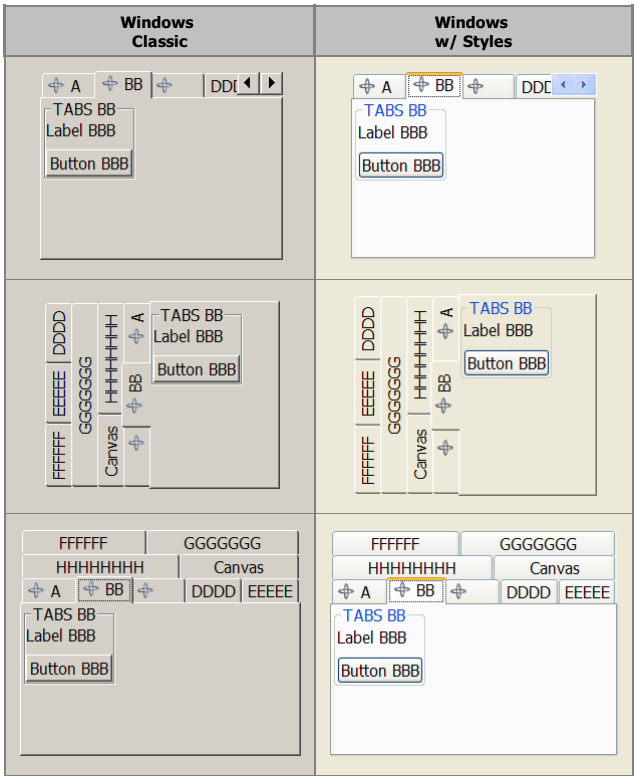
```
IupSetAttributeId(ih, "TABTITLE", 3, value) == IupSetAttribute(ih, "TABTITLE3", value)
```

But these functions are faster than the traditional functions because they do not need to parse the attribute name string and the application does not need to concatenate the attribute name with the id.

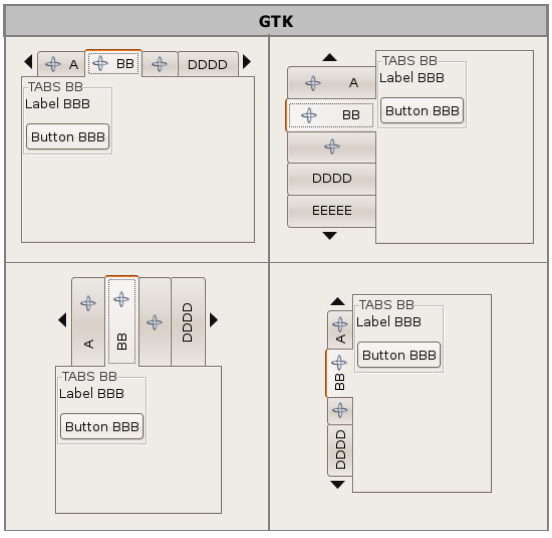
Examples

Browse for Example Files

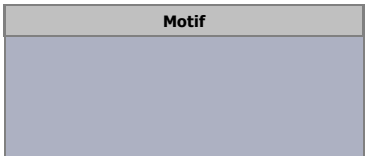
In Windows, the Visual Styles work only when TABTYPE is TOP.

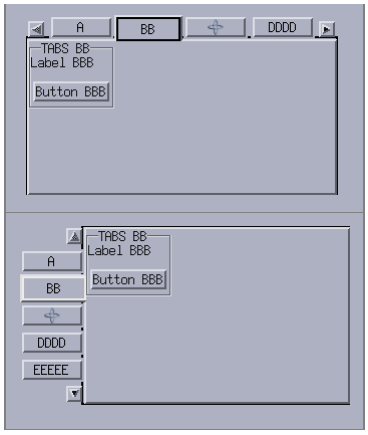


GTK is the only one that supports vertical text in the TOP configuration, but does not supports multiple lines of tab buttons.



Motif does not supports vertical text.





IupAnimatedLabel (since 3.17)

Creates an animated label interface element, which displays an image that is changed periodically.

It uses an animation that is simply an **IupUser** with several **IupImage** as children.

It inherits from IupLabel.

Creation

```
Ihandle* IupAnimatedLabel(Ihandle* animation); [in C]
iup.animatedlabel{animation: ihandle} -> (ih: ihandle) [in Lua]
animatedlabel(animation) [in LED]
```

animation: element that contains the list of images. It can be NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

All **IupLabel** attributes. The IMAGE attribute is periodically changed by a timer.

Additionally it defines the following non-inheritable attributes.

START (write-only): starts the animation. The value is ignored. By default the animation is stopped.

STOP (write-only): stops the animation. The value is ignored.

RUNNING (read-only): return YES if the animation is running.

FRAMETIME: The time between each frame. If the **IupUser** element has a FRAMETIME attribute it will be used to set the **IupAnimatedLabel** FRAMETIME attribute, but it can be overwritten later on.

FRAMECOUNT (read-only): number of frames in the animation. It is simply **IupGetChildCount** of the given **IupUser** element.

ANIMATION: the name of the element that contains the list of images. The value passed must be the name of an **IupUser** element with several **IupImage** as children. Use IupSetHandle or IupSetAttributeHandle to associate a child to a name. In Lua you can also use the element reference directly.

ANIMATION_HANDLE: same as ANIMATION but directly using the Ihandle* of the element.

Callbacks

All **IupLabel** callbacks. No label callbacks are used internally.

Notes

The **IupImageLib** contains a simple animation to show an indefinite progress called "**IUP_CircleProgressAnimation**".

The **IUP-IM** functions has two functions that can create an animation from image files called **IupLoadAnimation** and **IupLoadAnimationFrames**.

Examples

```
 label = IupAnimatedLabel(NULL);
IupSetAttribute(label, "ANIMATION", "IUP_CircleProgressAnimation");
IupSetAttribute(label, "START", "Yes");
```

Browse for Example Files

See Also

IupLabel, IupUser, IupImage, IupImageLib, IUP-IM.

IupButton

Creates an interface element that is a button. When selected, this element activates a function in the application. Its visual presentation can contain a text and/or an image.

Creation

```
Ihandle* IupButton(const char *title, const char *action); [in C]
iup.button{[title = title: string]} -> ih: ihandle [in Lua]
button(title, action) [in LED]
```

title: Text to be shown to the user. It can be NULL. It will set the TITLE attribute.
action: Name of the action generated when the button is selected. It can be NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

ALIGNMENT (non inheritable): horizontal and vertical alignment. Possible values: "ALEFT", "ACENTER" and "ARIGHT", combined to "ATOP", "ACENTER" and "ABOTTOM". Default: "ACENTER:ACENTER". Partial values are also accepted, like "ARIGHT" or ":ATOP", the other value will be used from the current alignment. In Motif, vertical alignment is restricted to "ACENTER". In GTK, horizontal alignment for multiple lines will align only the text block. (since 3.0)

BGCOLOR: Background color. If text and image are not defined, the button is configured to simply show a color, in this case set the button size because the natural size will be very small. In Windows, the BGCOLOR attribute is ignored if text or image is defined. Default: the global attribute DLGBGCOLOR. BGCOLOR is ignored when FLAT=YES because it will be used the background from the native parent.

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. In Windows the button will respect CANFOCUS differently to some other controls. Default: YES. (since 3.0)

FLAT (creation only): Hides the button borders until the mouse cursor enters the button area. Can be YES or NO. Default: NO.

FGCOLOR: Text color. Default: the global attribute DLGFGCOLOR.

IMAGE (non inheritable): Image name. If set before map defines the behavior of the button to contain an image. The natural size will be size of the image in pixels, plus the button borders. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. If TITLE is also defined and not empty both will be shown (except in Motif). (GTK 2.6)

IMINACTIVE (non inheritable): Image name of the element when inactive. If it is not defined then the IMAGE is used and the colors will be replaced by a modified version of the background color creating the disabled effect. GTK will also change the inactive image to look like other inactive objects. (GTK 2.6)

IMPRESS (non inheritable): Image name of the pressed button. If IMPRESS and IMAGE are defined, the button borders are not shown and not computed in natural size. When the button is clicked the pressed image does not offset. In Motif the button will lose its focus feedback also. (GTK 2.6)

IMPRESSBORDER (non inheritable): if enabled the button borders will be shown and computed even if IMPRESS is defined. Can be "YES" or "NO". Default: "NO".

IMAGEPOSITION (non inheritable): Position of the image relative to the text when both are displayed. Can be: LEFT, RIGHT, TOP, BOTTOM. Default: LEFT. (since 3.0) (GTK 2.10)

MARKUP [GTK only]: allows the title string to contains pango markup commands. Works only if a mnemonic is NOT defined in the title. Can be "YES" or "NO". Default: "NO".

PADDING: internal margin. Works just like the MARGIN attribute of the **IupHbox** and **IupVbox** containers, but uses a different name to avoid inheritance problems. Default value: "0x0". (since 3.0)

SPACING (creation only): defines the spacing between the image associated and the button's text. Default: "2".

TITLE (non inheritable): Button's text. If IMAGE is not defined before map, then the default behavior is to contain only a text. The button behavior can not be changed after map. The natural size will be larger enough to include all the text in the selected font, even using multiple lines, plus the button borders. The '\n' character is accepted for line change. The "&" character can be used to define a mnemonic, the next character will be used as key. Use "&&" to show the "&" character instead on defining a mnemonic. The button can be activated from any control in the dialog using the "Alt+key" combination. In old Motif versions (2.1) using a '\n' causes an invalid memory access inside Motif. (mnemonic support since 3.0)

ACTIVE, FONT, EXPAND, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

ACTION: Action generated when the button 1 (usually left) is selected. This callback is called only after the mouse is released and when it is released inside the button area.

```
int function(Ihandle* ih); [in C]
ih:action() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Returns: IUP_CLOSE will be processed.

BUTTON_CB: Action generated when any mouse button is pressed and when it is released. Both calls occur before the ACTION callback when button 1 is being used.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

Buttons with images and/or texts can not change its behavior after mapped. This is a creation dependency. But after creation the image can be changed for another image, and the text for another text.

Buttons are activated using Enter or Space keys.

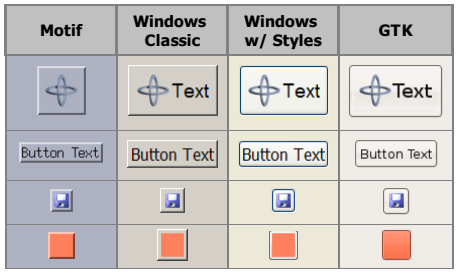
Buttons are not activated if the user clicks inside the button but moves the cursor and releases outside the button area. Also in Windows the highlight feedback when that happens is different if the button has CANFOCUS enabled or not.

Usually toolbar buttons have FLAT=Yes and CANFOCUS=NO.

Examples

Browse for Example Files

The buttons with image and text simultaneous have PADDING=5x5, the other buttons have no padding. The buttons with no text and BGCOLOR defined have their RASTERSIZE set.



See Also

IupImage, IupToggle, IupLabel

IupCalendar (since 3.17)

Creates a month calendar interface element, where the user can select a date.

GTK and Windows only. NOT available in Motif.

Creation

```
Ihandle* IupCalendar(void); [in C]
iup.calendar{} -> (ih: ihandle) [in Lua]
calendar() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

TODAY (read-only): Returns the date corresponding to today in VALUE format.

VALUE: the current date always in the format "year/month/day" ("%d/%d/%d" in C). Can be set to "TODAY". Default value is the today date.

WEEKNUMBERS: Shows the number of the week along the year. Default: NO.

Callbacks

VALUECHANGED_CB: Called after the value was interactively changed by the user.

```
int function(Ihandle *ih); [in C]
ih:valuechanged_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

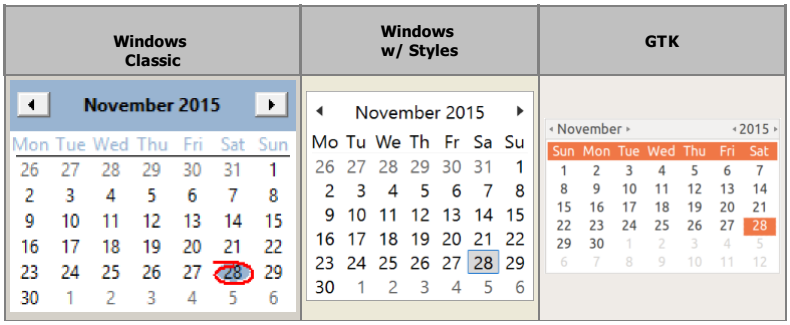
MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

In Windows, the view is changed when the month of year is clicked, so the user can select the month of the year or an year among years.

In GTK the today date is not marked in the calendar.

Examples



Browse for Example Files

See Also

IupDatePick.

IupCanvas

Creates an interface element that is a canvas - a working area for your application.

Creation

```
Ihandle* IupCanvas(const char *action); [in C]
iup.canvas{} -> (ih: ihandle) [in Lua]
canvas(action) [in LED]
```

action: Name of the action generated when the canvas needs to be redrawn. It can be NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BACKINGSTORE [Motif Only]: Controls the canvas backing store flag. The default value is "YES".

BGCOLOR: Background color. The background is painted only if the ACTION callback is not defined. If the callback is defined the application must draw all the canvas contents. In GTK or Motif if you set the ACTION callback after map then you should also set BGCOLOR to any value just after setting the callback or the first redraw will be lost. Default: "255 255 255".

BORDER (creation only): Shows a border around the canvas. Default: "YES".

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. In Windows the canvas will respect CANFOCUS differently to some other controls. Default: YES. (since 3.0)

CAIRO_CR [GTK Only] (non inheritable): Contains the "cairo_t*" parameter of the internal GTK callback. Valid only during the ACTION callback and onyl when using GTK version 3. (since 3.7)

CLIPRECT [Windows and GTK Only] (only during ACTION): Specifies a rectangle that has its region invalidated for painting, it could be used for clipping. Format: "%d %d %d %d"="x1 y1 x2 y2".

CURSOR (non inheritable): Defines a cursor for the canvas. The Windows SDK recommends that cursors and icons should be implemented as resources rather than created at run time.

EXPAND (non inheritable): The default value is "YES". The natural size is the size of 1 character.

DROPFILESTARGET [Windows and GTK Only] (non inheritable): Enable or disable the drop of files. Default: NO, but if DROPFILES_CB is defined when the element is mapped then it will be automatically enabled.

DRAWSIZE (non inheritable): The size of the drawing area in pixels. This size is also used in the RESIZE_CB callback.

Notice that the drawing area size is not the same as RASTERSIZE. The SCROLLBAR and BORDER attributes affect the size of the drawing area.

HDC_WMPAINT [Windows Only] (non inheritable): Contains the HDC created with the BeginPaint inside the WM_PAINT message. Valid only during the ACTION callback.

HWND [Windows Only] (non inheritable, read-only): Returns the Windows Window handle. Available in the Windows driver or in the GTK driver in Windows.

SCROLLBAR (creation only): Associates a horizontal and/or vertical scrollbar to the canvas. Default: "NO". The secondary attributes are all non inheritable.

DX: Size of the thumb in the horizontal scrollbar. Also the horizontal page size. Default: "0.1".
DY: Size of the thumb in the vertical scrollbar. Also the vertical page size. Default: "0.1".
POSX: Position of the thumb in the horizontal scrollbar. Default: "0.0".
POSY: Position of the thumb in the vertical scrollbar. Default: "0.0".
XMIN: Minimum value of the horizontal scrollbar. Default: "0.0".
XMAX: Maximum value of the horizontal scrollbar. Default: "1.0".
YMIN: Minimum value of the vertical scrollbar. Default: "0.0".
YMAX: Maximum value of the vertical scrollbar. Default: "1.0".
LINEX: The amount the thumb moves when an horizontal step is performed. Default: 1/10th of DX. (since 3.0)
LINEY: The amount the thumb moves when a vertical step is performed. Default: 1/10th of DY. (since 3.0)
XAUTOHIDE: When enabled, if DX >= XMAX-XMIN then the horizontal scrollbar is hidden. Default: "YES". (since 3.0)
YAUTOHIDE: When enabled, if DY >= YMAX-YMIN then the vertical scrollbar is hidden. Default: "YES". (since 3.0)

TOUCH [Windows 7 Only]: enable the multi-touch events processing. (Since 3.3)

XDISPLAY [UNIX Only](non inheritable, read-only): Returns the X-Windows Display. Available in the Motif driver or in the GTK driver in UNIX.

XWINDOW [UNIX Only](non inheritable, read-only): Returns the X-Windows Window (Drawable). Available in the Motif driver or in the GTK driver in UNIX.

ACTIVE, FONT, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Drag & Drop attributes and callbacks are supported.

Callbacks

ACTION: Action generated when the canvas needs to be redrawn.

```
int function(Ihandle *ih, float posx, float posy); [in C]
ih:action(posx, posy: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
posx: thumb position in the horizontal scrollbar. The POSX attribute value.
posy: thumb position in the vertical scrollbar. The POSY attribute value.

BUTTON_CB: Action generated when any mouse button is pressed or released.

DROPFILES_CB [Windows and GTK Only]: Action generated when one or more files are dropped in the element.

FOCUS_CB: Called when the canvas gets or looses the focus. It is called after the common callbacks GETFOCUS_CB and KILL_FOCUS_CB.

```
int function(Ihandle *ih, int focus); [in C]
ih:focus_cb(focus: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
focus: is non zero if the canvas is getting the focus, is zero if it is loosing the focus.

MOTION_CB: Action generated when the mouse is moved.

KEYPRESS_CB: Action generated when a key is pressed or released. It is called after the common callback K_ANY.

When the canvas has the focus, pressing the arrow keys may change the focus to another control in some systems. If your callback process the arrow keys, we recommend you to return IUP_IGNORE so it will not lose its focus.

RESIZE_CB: Action generated when the canvas size is changed.

SCROLL_CB: Called when the scrollbar is manipulated. (GTK 2.8) Also the POSX and POSY values will not be correctly updated for older GTK versions.

TOUCH_CB [Windows 7 Only]: Action generated when a touch event occurred. Multiple touch events will trigger several calls. Must set TOUCH=Yes to receive this event. (Since 3.3)

```
int function(Ihandle* ih, int id, int x, int y, char* state); [in C]
ih:touch_cb(id, x, y: number, state: string) -> (ret: number) [in Lua]
```

ih: identifies the element that activated the event.
id: identifies the touch point.
x, **y**: position in pixels, relative to the top-left corner of the canvas.
state: the touch point state. Can be: DOWN, MOVE or UP. If the point is a "primary" point then "-PRIMARY" is appended to the string.

Returns: IUP_CLOSE will be processed.

MULTITOUCH_CB [Windows 7 Only]: Action generated when multiple touch events occurred. Must set TOUCH=Yes to receive this event. (Since 3.3)

```
int function(Ihandle *ih, int count, int* pid, int* px, int* py, int* pstate) [in C]
ih:multitouch_cb(count: number, pid, px, py, pstate: table) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
count: Number of touch points in the array.
pid: Array of touch point ids.
px: Array of touch point x coordinates in pixels, relative to the top-left corner of the canvas.
py: Array of touch point y coordinates in pixels, relative to the top-left corner of the canvas.
pstate: Array of touch point states. Can be 'D' (DOWN), 'U' (UP) or 'M' (MOVE).

Returns: IUP_CLOSE will be processed.

WHEEL_CB: Action generated when the mouse wheel is rotated.

WOM_CB [Windows Only]: Action generated when an audio device receives an event.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Drag & Drop attributes and callbacks are supported.

Notes

Note that some keys might remove the focus from the canvas. To avoid this, return IGNORE in the K_ANY callback.

The mouse cursor position can be programmatically controlled using the global attribute CURSORPOS.

When the canvas is displayed for the first time, the callback call order is always:

```
MAP_CB()
RESIZE_CB()
ACTION()
```

When the canvas is resized the ACTION callback is always called after the RESIZE_CB callback.

Using with the CD library

When using the CD library to draw in a IupCanvas, you can use the callbacks to manage the canvas. The simplest way is to do:

```
MAP_CB    - calls cdCreateCanvas  (current size is not available yet)
UNMAP_CB  - calls cdKillCanvas
RESIZE_CB - Calling cdCanvasActivate and cdCanvasGetSize returns the same values as
            given by the callback parameters.
            Recalculate the drawing size, update the scrollbars if any.
ACTION - calls cdCanvasActivate
         then use CD primitives to draw the scene,
         finally calls cdCanvasFlush if using double buffer
SCROLL_CB - when using scrollbars,
            if this callback is defined the canvas must be manually redrawn,
            call yourself the action callback or call IupUpdate.
            In other words, if this callback is not defined
            the canvas is automatically redrawn.
```

Examples

Browse for Example Files

SCROLLBAR (creation only)

Associates a horizontal and/or vertical scrollbar to the element.

Value

"VERTICAL", "HORIZONTAL", "YES" (both) or "NO" (none).

Default: "NO"

Configuration Attributes (non inheritable)

DX: Size of the thumb in the horizontal scrollbar. Also the horizontal page size. Default: "0.1".

DY: Size of the thumb in the vertical scrollbar. Also the vertical page size. Default: "0.1".

POSX: Position of the thumb in the horizontal scrollbar. Default: "0.0".

POSY: Position of the thumb in the vertical scrollbar. Default: "0.0".

XMIN: Minimum value of the horizontal scrollbar. Default: "0.0".

XMAX: Maximum value of the horizontal scrollbar. Default: "1.0".

YMIN: Minimum value of the vertical scrollbar. Default: "0.0".

YMAX: Maximum value of the vertical scrollbar. Default: "1.0".

LINEX: The amount the thumb moves when an horizontal step is performed. Default: 1/10th of DX. (since 3.0)

LINEY: The amount the thumb moves when a vertical step is performed. Default: 1/10th of DY. (since 3.0)

XAUTOHIDE: When enabled, if DX >= XMAX-XMIN then the horizontal scrollbar is hidden. Default: "YES". (since 3.0)

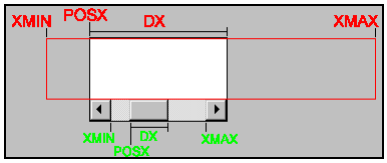
YAUTOHIDE: When enabled, if DY >= YMAX-YMIN then the vertical scrollbar is hidden. Default: "YES". (since 3.0)

XHIDDEN: returns if the scrollbar is hidden or not when XAUTOHIDE=Yes. (since 3.13)

YHIDDEN: returns if the scrollbar is hidden or not when YAUTOHIDE=Yes. (since 3.13)

Notes

The scrollbar allows you to create a virtual space associated to the element. In the image below, such space is marked in **red**, as well as the attributes that affect the composition of this space. In **green** you can see how these attributes are reflected on the scrollbar.



Hence you can clearly deduce that POSX is limited to XMIN and XMAX-DX, or **XMIN<=POSX<=XMAX-DX**.

Usually applications configure XMIN and XMAX to a region in World coordinates, and set DX to the canvas visible area in World coordinates. Since the canvas can have scrollbars and borders, its visible area in pixel coordinates can be easily obtained using the **DRAWSIZE** attribute.

IMPORTANT: the LINEX, XMAX and XMIN attributes are only updated in the scrollbar when the DX attribute is updated.

IMPORTANT: when working with a virtual space with integer coordinates, set XMAX to the integer size of the virtual space, NOT to "width-1", or the last pixel of the virtual space will never be visible. If you decide to let XMAX with the default value of 1.0 and to control only DX, then use the formula DX=visible_width/width.

IMPORTANT: When the virtual space has the same size as the canvas, i.e. when **DX >= XMAX-XMIN**, the scrollbar is automatically hidden if **XAUTOHIDE**=Yes. The width of the vertical scrollbar (the same as the height of the horizontal scrollbar) can be obtained using the SCROLLBARSIZE global attribute (since 3.9).

The same is valid for YMIN, YMAX, DY and POSY. But remember that the Y axis is oriented from top to bottom in IUP. So if you want to consider YMIN and YMAX as bottom-up oriented, then the actual YPOS must be obtained using **YMAX-DY-POSY**.

IMPORTANT: Changes in the scrollbar parameters do NOT generate ACTION nor SCROLL_CB callback events. If you need to update the canvas contents call your own action callback or call **IupUpdate**. But a change in the DX attribute may generate a RESIZE_CB callback event if XAUTOHIDE=Yes.

If you have to change the properties of the scrollbar (XMIN, XMAX and DX) but you want to keep the thumb still (if possible) in the same relative position, then you have to also recalculate its position (POSX) using the old position as reference to the new one. For example, you can convert it to a 0-1 interval and then scale to the new limits:

```
old_posx_relative = (old_posx - old_xmin)/(old_xmax - old_xmin)
posx = (xmax - xmin)*old_posx_relative + xmin
```

IupList, IupTree, and **IupText/IupMultiline** scrollbars are automatically managed and do NOT have the POS*, *MIN, *MAX and D* attributes.

When updating the virtual space size, or when the canvas is resized, if **XAUTOHIDE**=Yes then calculating the actual DX size can be very tricky. Here is a a helpful algorithm:

```
void scrollbar_update(Ihandle* ih, int view_width, int view_height)
{
  /* view_width and view_height is the virtual space size */
  /* here we assume XMIN=0, XMAX=1, YMIN=0, YMAX=1 */
  int elem_width, elem_height;
  int canvas_width, canvas_height;
  int scrollbar_size = IupGetInt(NULL, "SCROLLBARSIZE");
  int border = IupGetInt(ih, "BORDER");

  IupGetIntInt(ih, "RASTERSIZE", &elem_width, &elem_height);

  /* if view is greater than canvas in one direction,
     then it has scrollbars,
     but this affects the opposite direction */
  elem_width -= 2 * border;  /* remove BORDER (always size=1) */
  elem_height -= 2 * border;
  canvas_width = elem_width;
  canvas_height = elem_height;
  if (view_width > elem_width)  /* check for horizontal scrollbar */
    canvas_height -= scrollbar_size;  /* affect vertical size */
  if (view_height > elem_height)
    canvas_width -= scrollbar_size;
  if (view_width <= elem_width && view_width > canvas_width)  /* check if still has horizontal scrollbar */
    canvas_height -= scrollbar_size;
  if (view_height <= elem_height && view_height > canvas_height)
    canvas_width -= scrollbar_size;
  if (canvas_width < 0) canvas_width = 0;
  if (canvas_height < 0) canvas_height = 0;

  IupSetFloat(ih, "DX", (float)canvas_width / (float)view_width);
  IupSetFloat(ih, "DY", (float)canvas_height / (float)view_height);
}
```

Inside the canvas ACTION callback, the (x,y) offset for drawing is calculated as:

```
 int x, y, canvas_width, canvas_height;
float posy = IupGetFloat(ih, "POSY");
float posx = IupGetFloat(ih, "POSX");

IupGetIntInt(ih, "DRAWSIZE", &canvas_width, &canvas_height);

if (canvas_width < view_width)
  x = (int)floor(-posx*view_width);
else
  x = (canvas_width - view_width) / 2;  /* for example, center the view */

if (canvas_height < view_height)
{
  /* posy is top-bottom, CD and OpenGL are bottom-top.
     invert posy reference (YMAX-DY - POSY) */
  float dy = IupGetFloat(ih, "DY");
  posy = 1.0f - dy - posy;
  y = (int)floor(-posy*view_height);
}
else
  y = (canvas_height - view_height) / 2;  /* for example, center the view */
```

Call **scrollbar_update** from the RESIZE_CB callback and when you change the zoom factor that affects **view_width** or **view_height**.

Affects

IupList, IupMultiline, IupCanvas

See Also

POSX, XMIN, XMAX, DX, POSY, YMIN, YMAX, DY

DX

Size of the horizontal scrollbar's thumbnail in any unit.

Value

Any floating-point value greater than zero and smaller than the difference between XMAX and XMIN.

Default:: "0.1".

Notes

LINEX, XMAX and XMIN are only updated in the scrollbar when DX is updated.

When the canvas is visible, a change in DX can generate a redraw in the horizontal scrollbar on the screen. But it may generate a RESIZE_CB callback event if XAUTOHIDE=Yes.

A change in these values can affect the attribute POSX.

Affects

IupCanvas

See Also

SCROLLBAR

DY

Size of the vertical scrollbar's thumbnail in any unit.

Value

Any floating-point value greater than zero and smaller than the difference between YMAX and YMIN.

Default:: "0.1".

Notes

LINEY, YMAX and YMIN are only updated in the scrollbar when DY is updated.

When the canvas is visible, a change in DY can generate a redraw in the horizontal scrollbar on the screen. But it may generate a RESIZE_CB callback event if YAUTOHIDE=Yes.

A change in these values can affect the attribute POSY.

Affects

IupCanvas

See Also

SCROLLBAR

POSX

Thumbnail position in the horizontal scrollbar in any unit.

Value

Any floating-point value. Must be a value between XMIN and XMAX-DX.

Default: "0.0"

Notes

When the canvas is visible, a change in POSX can generate a redraw in the horizontal scrollbar on the screen, but will NOT generate a redraw of the canvas.

Affects

IupCanvas

See Also

SCROLLBAR

POSY

Thumbnail position in the vertical scrollbar in any unit.

Value

Any floating-point value. Must be a value between YMIN and YMAX-DY.

Default: "0.0"

Notes

When the canvas is visible, a change in POSY can generate a redraw in the vertical scrollbar on the screen, but will NOT generate a redraw of the canvas.

Affects

IupCanvas

See Also

SCROLLBAR

XMIN

Minimum value of the horizontal scrollbar, in any unit.

Value

Any floating-point value.

Default: "0.0"

Notes

A change in this value will only be effective after the attribute DX is changed.

Affects

IupCanvas

See Also

SCROLLBAR

XMAX

Maximum value of the horizontal scrollbar, in any unit.

Value

Any floating-point value.

Default: "1.0"

Notes

A change in this value will only be effective after the attribute DX is changed.

Affects

IupCanvas

See Also

SCROLLBAR

YMIN

Minimum value of the vertical scrollbar, in any unit.

Value

Any floating-point value.

Default: "0.0"

Notes

A change in this value will only be effective after the attribute DY is changed.

Affects

IupCanvas

See Also

SCROLLBAR

YMAX

Maximum value of the vertical scrollbar, in any unit.

Value

Any floating-point value.

Default: "1.0"

Notes

A change in this value will only be effective after the attribute DY is changed.

Affects

IupCanvas

See Also

SCROLLBAR

BUTTON_CB

Action generated when a mouse button is pressed or released.

Callback

```
int function(Ihandle* ih, int button, int pressed, int x, int y, char* status); [in C]
ih:button_cb(button, pressed, x, y: number, status: string) -> (ret: number) [in Lua]
```

ih: identifies the element that activated the event.
button: identifies the activated mouse button:

IUP_BUTTON1 - left mouse button (button 1);
IUP_BUTTON2 - middle mouse button (button 2);
IUP_BUTTON3 - right mouse button (button 3).

pressed: indicates the state of the button:

0 - mouse button was released;
1 - mouse button was pressed.

x, **y**: position in the canvas where the event has occurred, in pixels.
status: status of the mouse buttons and some keyboard keys at the moment the event is generated. The following macros must be used for verification:

iup_isshift(status)
iup_iscontrol(status)
iup_isbutton1(status)
iup_isbutton2(status)
iup_isbutton3(status)
iup_isbutton4(status)
iup_isbutton5(status)
iup_isdouble(status)
iup_isalt(status)
iup_issys(status)

They return 1 if the respective key or button is pressed, and 0 otherwise. These macros are also available in Lua, returning a boolean.

Returns: IUP_CLOSE will be processed. On some controls if IUP_IGNORE is returned the action is ignored (this is system dependent).

Notes

This callback can be used to customize a button behavior. For a standard button behavior use the ACTION callback of the **IupButton**.

For a single click the callback is called twice, one for pressed=1 and one for pressed=0. Only after both calls the ACTION callback is called. In Windows, if a dialog is shown or popup in any situation there could be unpredictable results because the native system still has processing to be done even after the callback is called.

A double click is preceded by two single clicks, one for pressed=1 and one for pressed=0, and followed by a press=0, all three without the double click flag set. In GTK, it is preceded by an additional two single clicks sequence. For example, for one double click all the following calls are made:

```
BUTTON_CB(but=1 (1), x=154, y=83 [  1       ])
BUTTON_CB(but=1 (0), x=154, y=83 [  1       ])
    BUTTON_CB(but=1 (1), x=154, y=83 [  1       ])      (in GTK only)
    BUTTON_CB(but=1 (0), x=154, y=83 [  1       ])      (in GTK only)
BUTTON_CB(but=1 (1), x=154, y=83 [  1  D    ])
BUTTON_CB(but=1 (0), x=154, y=83 [  1       ])
```

Affects

IupCanvas, IupButton, IupText, IupList, IupGLCanvas

MOTION_CB

Action generated when the mouse moves.

Callback

```
int function(Ihandle *ih, int x, int y, char *status); [in C]
ih:motion_cb(x, y: number, status: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
x, **y**: position in the canvas where the event has occurred, in pixels.
status: status of mouse buttons and certain keyboard keys at the moment the event was generated. The same macros used for BUTTON_CB can be used for this status.

Affects

IupCanvas, IupGLCanvas

KEYPRESS_CB

Action generated when a key is pressed or released. If the key is pressed and held several calls will occur. It is called after the callback **K_ANY** is processed.

Callback

```
int function(Ihandle *ih, int c, int press); [in C]
ih:keypress_cb(c, press: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
c: identifier of typed key. Please refer to the Keyboard Codes table for a list of possible values.
press: 1 is the user pressed the key or 0 otherwise.

Returns: If IUP_IGNORE is returned the key is ignored by the system. IUP_CLOSE will be processed.

Affects

IupCanvas

SCROLL_CB

Called when some manipulation is made to the scrollbar. The canvas is automatically redrawn only if this callback is NOT defined.

(GTK 2.8) Also the POSX and POSY values will not be correctly updated for older GTK versions.

In Ubuntu, when liboverlay-scrollbar is enabled (the new tiny auto-hide scrollbar) only the IUP_SBPOSV and IUP_SBPOSH codes are used.

Callback

```
int function(Ihandle *ih, int op, float posx, float posy); [in C]
ih:scroll_cb(op, posx, posy: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
op: indicates the operation performed on the scrollbar.

If the manipulation was made on the vertical scrollbar, it can have the following values:

```
IUP_SBUP - line up
IUP_SBDN - line down
IUP_SBPGUP - page up
IUP_SBPGDN - page down
IUP_SBPOSV - vertical positioning
IUP_SBDRAGV - vertical drag
```

If it was on the horizontal scrollbar, the following values are valid:

```
IUP_SBLEFT - column left
IUP_SBRIGHT - column right
IUP_SBPGLEFT - page left
IUP_SBPGRIGHT - page right
IUP_SBPOSH - horizontal positioning
IUP_SBDRAGH - horizontal drag
```

posx, **posy**: the same as the **ACTION** canvas callback (corresponding to the values of attributes POSX and POSY).

Notes

IUP_SBDRAGH and IUP_SBDRAGV are not supported in GTK.

Affects

IupCanvas, IupGLCanvas, SCROLLBAR

WHEEL_CB

Action generated when the mouse wheel is rotated. If this callback is not defined the wheel will automatically scroll the canvas in the vertical direction by some lines, the SCROLL_CB callback if defined will be called with the IUP_SBDRAGV operation.

Callback

```
int function(Ihandle *ih, float delta, int x, int y, char *status); [in C]
ih:wheel_cb(delta, x, y: number, status: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
delta: the amount the wheel was rotated in notches.
x, **y**: position in the canvas where the event has occurred, in pixels.
status: status of mouse buttons and certain keyboard keys at the moment the event was generated. The same macros used for BUTTON_CB can be used for this status.

Notes

In Motif and GTK delta is always 1 or -1. In Windows is some situations delta can reach the value of two. In the future with more precise wheels this increment can be changed.

Affects

IupCanvas, IupGLCanvas

WOM_CB

Action generated when an audio device receives an event.

[Windows Only]

Callback

```
int function(Ihandle *ih, int state); [in C]
ih:wom_cb(state: number) -> (ret: number) [in Lua]
```

ih: identifies the element that activated the event.
state: can be opening=1, done=0, or closing=-1.

Notes

This callback is used to syncronize video playback with audio. It is sent when the audio device:

Message	Description
opening	is opened by using the **waveOutOpen** function.
done	is finished with a data block sent by using the **waveOutWrite** function.
closing	is closed by using the **waveOutClose** function.

You must use the HWND attribute when calling **waveOutOpen** in the *dwCallback* parameter and set *fdwOpen* to CALLBACK_WINDOW.

Affects

IupDialog, IupCanvas, IupGLCanvas

IupDatePick (since 3.17)

Creates a date editing interface element, which can displays a calendar for selecting a date.

In Windows is a native element. In GTK and Motif is a custom element. In Motif is not capable of displaying the calendar.

Creation

```
Ihandle* IupDatePick(); [in C]
iup.datepick{} -> (ih: ihandle) [in Lua]
datepick() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

CALENDARWEEKNUMBERS: Shows the number of the week along the year in the calendar. Default: NO.

FORMAT [Windows Only]: Flexible format for the date in Windows. For more information see "About Date and Time Picker Control" in the Windows SDK. The Windows control was configured to display date only without any time options. Default: "d'/'M'/'yyyy". See Noted bellow.

MONTHSHORTNAMES [Windows Only]: Month display will use a short name instead of numbers. Must be set before ORDER. Default: NO. Names will be in the language of the system.

ORDER: Day, month and year order. Can be any combination of "D", "M" and "Y" without repetition, and with all three letters. It will set the FORMAT attribute in Windows. It will NOT affect the VALUE attribute order. Default: "DMY".

SEPARATOR: Separator between day, month and year. Must be set before ORDER in Windows. Default: "/".

TODAY (read-only): Returns the date corresponding to today in VALUE format.

VALUE: the current date always in the format "year/month/day" ("%d/%d/%d" in C). Can be set to "TODAY". Default value is the today date.

ZEROPRECED: Day and month numbers will be preceded by a zero. Must be set before ORDER in Windows. Default: No.

Callbacks

VALUECHANGED_CB: Called after the value was interactively changed by the user.

```
int function(Ihandle *ih); [in C]
ih:valuechanged_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

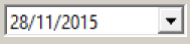
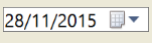
Notes

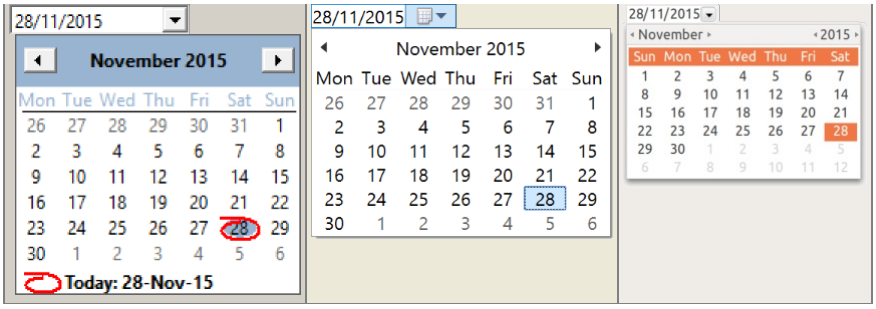
In Windows, when the user navigates to other pages in the calendar the date is not changed until the user actually selects a day.

In Windows, FORMAT can have the following values, but other text in the format string must be enclosed in single quotes:

Element	Description
"d"	The one- or two-digit day. (**default**)
"dd"	The two-digit day. Single-digit day values are preceded by a zero. (Set when ZEROPRECED=Yes)
"ddd"	The three-character weekday abbreviation.
"dddd"	The full weekday name.
"M"	The one- or two-digit month number. (**default**)
"MM"	The two-digit month number. Single-digit values are preceded by a zero. (Set when ZEROPRECED=Yes)
"MMM"	The three-character month abbreviation. (Set when MONTHSHORTNAMES=Yes)
"MMMM"	The full month name.
"yy"	The last two digits of the year (that is, 1996 would be displayed as "96"). (Not recommended)
"yyyy"	The full year (that is, 1996 would be displayed as "1996"). (**default**)

Examples

Windows Classic	Windows w/ Styles	GTK
		



Browse for Example Files

See Also

IupCalendar.

IupFlatButton (since 3.15)

Creates an interface element that is a button, but it does not have native decorations. When selected, this element activates a function in the application. Its visual presentation can contain a text and/or an image.

It behaves just like an IupButton, but since it is not a native control it has more flexibility for additional options. It can also behave like an IupToggle (without the checkmark).

It inherits from IupCanvas.

Creation

```
Ihandle* IupFlatButton(const char *title); [in C]
iup.flatbutton{[title = title: string]} -> ih: ihandle [in Lua]
flatbutton(title) [in LED]
```

title: Text to be shown to the user. It can be NULL. It will set the TITLE attribute.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

Inherits all attributes and callbacks of the IupCanvas, but redefines a few attributes.

ALIGNMENT (non inheritable): horizontal and vertical alignment. Possible values: "ALEFT", "ACENTER" and "ARIGHT", combined to "ATOP", "ACENTER" and "ABOTTOM". Default: "ACENTER:ACENTER". Partial values are also accepted, like "ARIGHT" or ":ATOP", the other value will be used from the current alignment.

BACKIMAGE (non inheritable): image name to be used as background. It will be zoomed to fill the background (it does not includes the border). Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage.

BACKIMAGEHIGHLIGHT (non inheritable): background image name of the element in highlight state. If it is not defined then the BACKIMAGE is used.

BACKIMAGEINACTIVE (non inheritable): background image name of the element when inactive. If it is not defined then the BACKIMAGE is used and its colors will be replaced by a modified version creating the disabled effect.

BACKIMAGEPRESS (non inheritable): background image name of the element in pressed state. If it is not defined then the BACKIMAGE is used.

BGCOLOR: Background color. If text and image are not defined, the button is configured to simply show a color, in this case set the button size because the natural size will be very small. Default: it will use used the background from the native parent the global attribute DLGBGCOLOR. BGCOLOR is ignored when FLAT=YES because it will be used the background from the native parent.

BORDER (creation only): the default value is "NO". This is the **IupCanvas** border.

BORDERCOLOR: color used for borders. Default: "50 150 255". This is for the **IupFlatButton** drawn border.

BORDERWIDTH: line width used for borders. Default: "1". Any borders can be hidden by simply setting this value to 0. This is for the **IupFlatButton** drawn border.

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. In Windows the button will respect CANFOCUS in opposite to the other controls. Default: YES.

EXPAND (non inheritable): The default value is "NO".

FGCOLOR: Text color. Default: the global attribute DLGFGCOLOR.

FITTOBACKIMAGE (non inheritable): enable the natural size to be computed from the BACKIMAGE. If BACKIMAGE is not defined will be ignored. When set to Yes it will set BORDERWIDTH to 0. Can be Yes or No. Default: No.

FRONTIMAGE (non inheritable): image name to be used as foreground. It will be zoomed to fill the foreground (it does not includes the border). The foreground has the same are as the background, but it is drawn at last. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage.

FRONTIMAGEHIGHLIGHT (non inheritable): foreground image name of the element in highlight state. If it is not defined then the FRONTIMAGE is used.

FRONTIMAGEINACTIVE (non inheritable): foreground image name of the element when inactive. If it is not defined then the FRONTIMAGE is used and its colors will be replaced by a modified version creating the disabled effect.

FRONTIMAGEPRESS (non inheritable): foreground image name of the element in pressed state. If it is not defined then the FRONTIMAGE is used.

HLCOLOR: color used to indicate a highlight state. Default: "200 225 245".

PSCOLOR: color used to indicate a press state. Default: "150 200 235".

IMAGE (non inheritable): Image name. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage.

IMAGEHIGHLIGHT (non inheritable): Image name of the element in highlight state. If it is not defined then the IMAGE is used.

IMAGEINACTIVE (non inheritable): Image name of the element when inactive. If it is not defined then the IMAGE is used and its colors will be replaced by a modified version creating the disabled effect.

IMAGEPRESS (non inheritable): Image name of the element in pressed state. If it is not defined then the IMAGE is used.

IMAGEPOSITION (non inheritable): Position of the image relative to the text when both are displayed. Can be: LEFT, RIGHT, TOP, BOTTOM. Default: LEFT.

PADDING: internal margin. Works just like the MARGIN attribute of the **IupHbox** and **IupVbox** containers, but uses a different name to avoid inheritance problems. Default value: "0x0".

RADIO (read-only): returns if the toggle is inside a radio. Can be "YES" or "NO". Valid only after the element is mapped and TOGGLE=Yes, before returns NULL.

SPACING (non inheritable): defines the spacing between the image associated and the button's text. Default: "2".

TITLE (non inheritable): Label's text. The '\n' character is accepted for line change.

TOGGLE: enabled the toggle behavior. Default: NO.

VALUE (non inheritable): Toggle's state. Values can be "ON", "OFF" or "TOGGLE". Default: "OFF". The TOGGLE option will invert the current state. Valid only when TOGGLE=Yes. Can only be set to Yes for a toggle inside a radio, it will automatically set to OFF the previous toggle that was ON.

ACTIVE, FONT, EXPAND, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

Inherits all callbacks of the IupCanvas, but redefines a few of them. Including ACTION, BUTTON_CB, FOCUS_CB, LEAVEWINDOW_CB, and ENTERWINDOW_CB. To allow the application to use those callbacks the same callbacks are exported with the "FLAT_" prefix using the same parameters, except the FLAT_ACTION callback that now mimics the **IupButton** ACTION. They are all called before the internal callbacks and if they return IUP_IGNORE the internal callbacks are not processed.

FLAT_ACTION: Action generated when the button 1 (usually left) is selected. This callback is called only after the mouse is released and when it is released inside the button area.

```
int function(Ihandle* ih); [in C]
ih:action() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Returns: IUP_CLOSE will be processed.

VALUECHANGED_CB: Called after the value was interactively changed by the user. Called only when TOGGLE=Yes. Called after the ACTION callback, but under the same context.

```
int function(Ihandle *ih); [in C]
ih:valuechanged_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

The **IupFlatButton** can contain text and image simultaneously.

The natural size will be a combination of the size of the image and the title, if any, plus PADDING and SPACING (if both image and title are present).

Borders are drawn only when the button is highlighted reproducing the behavior of the **IupButton** when FLAT=Yes.

Buttons are activated using Enter or Space keys.

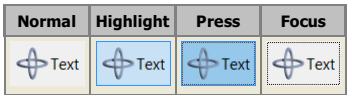
When TOGGLE=Yes, To build a set of mutual exclusive toggles, insert them in a **IupRadio** container. They must be inserted before creation, and their behavior can not be changed.

When TOGGLE=Yes, the button that is a child of an **IupRadio** automatically receives a name when its is mapped into the native system. (since 3.16)

Examples

Browse for Example Files

The sample buttons have PADDING=5x5.



See Also

IupImage, IupButton, IupToggle, IupLabel

Browse for Example Files

See Also

IupImage, IupButton.

See Also

IupLabel, IupHelp.

See Also

IupListDialog, Iuptext

IupProgressBar (since 3.0)

Creates a progress bar control. Shows a percent value that can be updated to simulate a progression.

It is similar of **IupGauge**, but uses native controls internally. Also does not have support for text inside the bar.

Creation

```
Ihandle* IupProgressBar(void); [in C]
iup.progressbar{} -> (ih: ihandle) [in Lua]
progressbar() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BGCOLOR [Windows Classic and Motif only]: controls the background color. Default: the global attribute DLGBGCOLOR.

DASHED (creation only in Windows) [Windows and GTK only]: Changes the style of the progress bar for a dashed pattern. Default is "NO".

FGCOLOR [Windows Classic and Motif only]: Controls the bar color. Default: the global attribute DLGFGCOLOR.

MARQUEE (creation): displays an undefined state. Default: NO. You can set the attribute after map but only to start or stop the animation. In Windows it will work only if using Visual Styles.

MAX (non inheritable): Contains the maximum value. Default is "1". The control display is not updated, must set VALUE attribute to update.

MIN (non inheritable): Contains the minimum value. Default is "0". The control display is not updated, must set VALUE attribute to update.

ORIENTATION (creation only): can be "VERTICAL" or "HORIZONTAL". Default: "HORIZONTAL". Horizontal goes from left to right, and vertical from bottom to top.

RASTERSIZE: The initial size is defined as "200x30". Set to NULL to allow the use of smaller values in the layout computation.

VALUE (non inheritable): Contains a number between "MIN" and "MAX", controlling the current position.

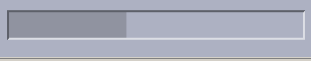
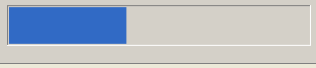
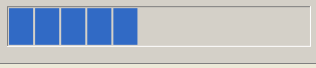
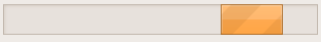
ACTIVE, EXPAND, FONT, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, ZORDER, VISIBLE: also accepted.

Callbacks

MAP_CB, UNMAP_CB, DESTROY_CB: common callbacks are supported.

Examples

Browse for Example Files

	DASHED=NO	DASHED=YES	MARQUEE=YES
Motif		(same as DASHED=NO)	
Windows Classic			(same as DASHED)
Windows w/ Styles	(same as DASHED=YES)		
Windows Vista		(same as DASHED=NO)	
GTK			

See Also

IupGauge

Notes

The spinbox can be created with no elements and be dynamic filled using IupAppend or IupInsert.

Examples

```
Ihandle* spinbox = IupSpinbox(IupText(NULL));
```

See Also

IupText, IupVbox, IupHbox, IupButton

IupText

Creates an editable text field.

Creation

```
Ihandle* IupText(const char *action); [in C]
iup.text{} -> (ih: ihandle) [in Lua]
text(action) [in LED]
```

action: name of the action generated when the user types something. It can be NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

ALIGNMENT [Windows and GTK Only] (non inheritable): text alignment. Possible values: "ALEFT", "ARIGHT", "ACENTER". Default: "ALEFT". In Motif, text is always left aligned.

APPEND (write-only): Inserts a text at the end of the current text. In the Multiline, if APPENDNEWLINE=YES, a "\n" character will be automatically inserted before the appended text if the current text is not empty(APPENDNEWLINE default is YES). Ignored if set before map.

BGCOLOR: Background color of the text. Default: the global attribute TXTBGCOLOR.

BORDER (creation only): Shows a border around the text. Default: "YES".

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. In Windows the control will still get the focus when clicked. Default: YES. (since 3.0)

CARET (non inheritable): **Character** position of the insertion point. Its format depends in MULTILINE=YES. The first position, **lin** or **col**, is "1".

For multiple lines: a string with the "**lin**,**col**" format, where **lin** and **col** are integer numbers corresponding to the caret's position.

For single line: a string in the "**col**" format, where **col** is an integer number corresponding to the caret's position.

When **lin** is greater than the number of lines, the caret is placed at the last line. When **col** is greater than the number of characters in the given line, the caret is placed after the last character of the line.

If the caret is not visible the text is scrolled to make it visible.

In Windows, if the element does not have the focus the returned value is the position of the first character of the current selection. The caret is only displayed if the element has the keyboard focus, but its position can be changed even if not visible. When changed it will also change the selection but the text will be scrolled only when it receives the focus.

See the Notes below if using UTF-8 strings in GTK.

CARETPOS (non inheritable): Also the **character** position of the insertion point, but using a zero based character unique index "pos". Useful for indexing the VALUE string. See the Notes below if using UTF-8 strings in GTK. (since 3.0)

CLIPBOARD (write-only): clear, cut, copy or paste the selection to or from the clipboard. Values: "CLEAR", "CUT", "COPY" or "PASTE". In Windows UNDO is also available, and REDO is available when FORMATTING=YES. (since 3.0)

COUNT (read-only): returns the number of **characters** in the text, including the line breaks. (since 3.5)

CUEBANNER [Windows Only] (non inheritable): a text that is displayed when there is no text at the control. It works as a textual cue, or tip to prompt the user for input. Valid only for MULTILINE=NO, and it is not available for Windows 2000. (since 3.0)

DROPFILESTARGET [Windows and GTK Only] (non inheritable): Enable or disable the drop of files. Default: NO, but if DROPFILES_CB is defined when the element is mapped then it will be automatically enabled. (since 3.0)

FGCOLOR: Text color. Default: the global attribute TXTFGCOLOR.

FILTER [Windows Only] (non inheritable): allows a custom filter to process the characters: Can be LOWERCASE, UPPERCASE or NUMBER (only numbers allowed). (since 3.0)

FORMATTING [Windows and GTK Only] (non inheritable): When enabled allow the use of text formatting attributes. In GTK is always enabled, but only when MULTILINE=YES. Default: NO. (since 3.0)

INSERT (write-only): Inserts a text in the caret's position, also replaces the current selection if any. Ignored if set before map.

LINECOUNT (read-only): returns the number of lines in the text. When MULTILINE=NO returns always "1". (since 3.5)

LINEVALUE (read-only): returns the text of the line where the caret is. It does not include the "\n" character. When MULTILINE=NO returns the same as VALUE. (since 3.5)

MASK (non inheritable): Defines a mask that will filter interactive text input.

MULTILINE (creation only) (non inheritable): allows the edition of multiple lines. In single line mode some characters are invalid, like "\t", "\r" and "\n". Default: NO. When set to Yes will also reset the SCROLLBAR attribute to Yes.

NC: Maximum number of **characters** allowed for keyboard input, larger text can still be set using attributes. The maximum value is the limit of the VALUE attribute. The "0" value is the same as maximum. Default: maximum.

NOHIDESEL [Windows Only]: do not hide the selection when the control loses its focus. Default: Yes. (since 3.16)

OVERWRITE [Windows and GTK Only] (non inheritable): turns the overwrite mode ON or OFF. Works only when FORMATTING=YES. (since 3.0)

PADDING: internal margin. Works just like the MARGIN attribute of the **IupHbox** and **IupVbox** containers, but uses a different name to avoid inheritance problems. Default value: "0x0". In Windows, only the horizontal value is used. (since 3.0) (GTK 2.10 for single line)

PASSWORD (creation only) [Windows and GTK Only] (non inheritable): Hide the typed character using an "*". Default: "NO".

READONLY: Allows the user only to read the contents, without changing it. Restricts keyboard input only, text value can still be changed using attributes. Navigation keys are still available. Possible values: "YES", "NO". Default: NO.

SCROLLBAR (creation only): Valid only when MULTILINE=YES. Associates an automatic horizontal and/or vertical scrollbar to the multiline. Can be: "VERTICAL", "HORIZONTAL", "YES" (both) or "NO" (none). Default: "YES". For all systems, when SCROLLBAR!=NO the natural size will always include its size even if the native system hides the scrollbar. If **AUTOHIDE**=YES scrollbars are visible only if they are necessary, by default AUTOHIDE=NO. In Windows when FORMATTING=NO, AUTOHIDE is not supported. In Motif AUTOHIDE is not supported.

SCROLLTO (non inheritable, write only): Scroll the text to make the given **character** position visible. It uses the same format and reference of the CARET attribute ("lin:col" or "col" starting at 1). In Windows, when FORMATTING=Yes "col" is ignored. (since 3.0)

SCROLLTOPOS (non inheritable, write only): Scroll the text to make the given **character** position visible. It uses the same format and reference of the CARETPOS attribute ("pos" starting at 0). (since 3.0)

SELECTEDTEXT (non inheritable): Selection text. Returns NULL if there is no selection. When changed replaces the current selection. Similar to INSERT, but does nothing if there is no selection.

SELECTION (non inheritable): Selection interval in **characters**. Returns NULL if there is no selection. Its format depends in MULTILINE=YES. The first position, **lin** or **col**, is "1".

For multiple lines: a string in the "**lin1**,**col1**:**lin2**,**col2**" format, where **lin1**, **col1**, **lin2** and **col2** are integer numbers corresponding to the selection's interval. **col2** correspond to the character after the last selected character.

For single line: a string in the "**col1**:**col2**" format, where **col1** and **col2** are integer numbers corresponding to the selection's interval. **col2** correspond to the character after the last selected character.

In Windows, when changing the selection the caret position is also changed.

The values ALL and NONE are also accepted independently of MULTILINE (since 3.0).

See the Notes below if using UTF-8 strings in GTK.

SELECTIONPOS (non inheritable): Same as SELECTION but using a zero based **character** index "**pos1**:**pos2**". Useful for indexing the VALUE string. The values ALL and NONE are also accepted. See the Notes below if using UTF-8 strings in GTK. (since 3.0)

SIZE (non inheritable): Since the contents can be changed by the user, the **Natural Size** is not affected by the text contents (since 3.0). Use VISIBLECOLUMNS and VISIBLELINES to control the **Natural Size**.

SPIN (non inheritable, creation only): enables a spin control attached to the element. Default: NO. The spin increments and decrements an integer number. The editing in the element is still available. (since 3.0)

SPINVALUE (non inheritable): the current value of the spin. The value is limited to the minimum and maximum values.
SPINMAX (non inheritable): the maximum value. Default: 100.
SPINMIN (non inheritable): the minimum value. Default: 0.
SPININC (non inheritable): the increment value. Default: 1.
SPINALIGN (creation only): the position of the spin. Can be LEFT or RIGHT. Default: RIGHT. In GTK is always RIGHT.
SPINWRAP (creation only): if the position reach a limit it continues from the opposite limit. Default: NO.
SPINAUTO (creation only): enables the automatic update of the text contents. Default: YES. Use SPINAUTO=NO and the VALUE attribute during SPIN_CB to control the text contents when the spin is incremented.

In Windows, the increment is multiplied by 5 after 2 seconds and multiplied by 20 after 5 seconds of a spin button pressed. In GTK, the increment change is progressively accelerated when a spin button is pressed.

TABSIZE [Windows and GTK Only]: Valid only when MULTILINE=YES. Controls the number of characters for a tab stop. Default: 8.

VALUE (non inheritable): Text entered by the user. The '\n' character indicates a new line, valid only when MULTILINE=YES. After the element is mapped and if there is no text will return the empty string "".

VALUEMASKED (non inheritable) (write-only): sets VALUE but first checks if it is validated by MASK. If not does nothing. (since 3.4)

VISIBLECOLUMNS: Defines the number of visible columns for the **Natural Size**, this means that will act also as minimum number of visible columns. It uses a wider character size then the one used for the SIZE attribute so strings will fit better without the need of extra columns. As for SIZE you can set to NULL after map to use it as an initial value. Default: 5 (since 3.0)

VISIBLELINES: When MULTILINE=YES defines the number of visible lines for the **Natural Size**, this means that will act also as minimum number of visible lines. As for SIZE you can set to NULL after map to use it as an initial value. Default: 1 (since 3.0)

WORDWRAP (creation only): Valid only when MULTILINE=YES. If enabled will force a word wrap of lines that are greater than the with of the control, and the horizontal scrollbar will be removed. Default: NO.

ACTIVE, FONT, EXPAND, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Drag & Drop attributes are supported. See Notes bellow.

Callbacks

ACTION: Action generated when the text is edited, but before its value is actually changed. Can be generated when using the keyboard, undo system or from the clipboard.

```
int function(Ihandle *ih, int c, char *new_value); [in C]
ih:action(c: number, new_value: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
c: valid alpha numeric character or 0.
new_value: Represents the new text value.

Returns: IUP_CLOSE will be processed, but the change will be ignored. If IUP_IGNORE, the system will ignore the new value. If **c** is valid and returns a valid alpha numeric character, this new character will be used instead. The VALUE attribute can be changed only if IUP_IGNORE is returned.

BUTTON_CB: Action generated when any mouse button is pressed or released. Use IupConvertXYToPos to convert (x,y) coordinates in character positioning. (since 3.0)

CARET_CB: Action generated when the caret/cursor position is changed.

```
int function(Ihandle *ih, int lin, int col, int pos); [in C]
ih:caret_cb(lin, col, pos: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, col: line and column number (start at 1).
pos: 0 based character position.

For single line controls **lin** is always 1, and **pos** is always "**col**-1".

DROPFILES_CB [Windows and GTK Only]: Action generated when one or more files are dropped in the element. (since 3.0)

MOTION_CB: Action generated when the mouse is moved. Use IupConvertXYToPos to convert (x,y) coordinates in character positioning. (since 3.0)

SPIN_CB: Action generated when a spin button is pressed. Valid only when SPIN=YES. When this callback is called the ACTION callback is not called. The VALUE attribute can be changed during this callback only if SPINAUTO=NO. (since 3.0)

```
int function(Ihandle *ih, int pos); [in C]
ih:spin_cb(pos: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
pos: the value of the spin (after it was incremented).

Returns: IUP_IGNORE is processed in Windows and Motif.

VALUECHANGED_CB: Called after the value was interactively changed by the user. (since 3.0)

```
int function(Ihandle *ih); [in C]
ih:valuechanged_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Drag & Drop callbacks are supported. See Notes bellow.

Auxiliary Functions

```
void IupTextConvertLinColToPos(Ihandle* ih, int lin, int col, int *pos); [in C]
iup.TextConvertLinColToPos(ih: ihandle, lin, col: number) -> pos: number [in Lua]
```

Converts a (lin, col) character positioning into an absolute position. lin and col starts at 1, pos starts at 0. For single line controls **pos** is always "**col**-1". (since 3.0)

```
void IupTextConvertPosToLinCol(Ihandle* ih, int pos, int *lin, int *col); [in C]
iup.TextConvertPosToLinCol(ih: ihandle, pos: number) -> lin, col: number [in Lua]
```

Converts an absolute position into a (lin, col) character positioning. lin and col starts at 1, pos starts at 0. For single line controls **lin** is always 1, and **col** is always "**pos**+1". (since 3.0)

Notes

When MULTILINE=YES the Enter key will add a new line, and the Tab key will insert a Tab. So the "DEFAULTENTER" button will not be processed when the element has the keyboard focus, also to change focus to the next element press <Ctrl>+<Tab>.

In Windows, if you press a Ctrl+key combination that is not supported by the control, then a beep is sound.

When using UTF-8 strings in GTK be aware that all attributes are indexed by characters, NOT by byte index, because some characters in UTF-8 can use more than one byte. This also applies to Windows if FORMATTING=YES depending on the Windows codepage (for example East Asian codepage where some characters take two bytes).

Internal Drag&Drop support is enabled by default. But in Windows the internal Drag&Drop is enabled only if FORMATTING=YES. In GTK the internal Drag&Drop can NOT be disabled, so it will conflict with the Drag & Drop attributes and callbacks.

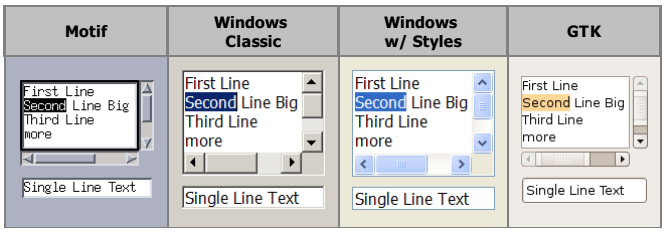
Navigation, Selection and Clipboard Keys

Here is a list of the common keys for all drivers. Other keys are available depending on the driver.

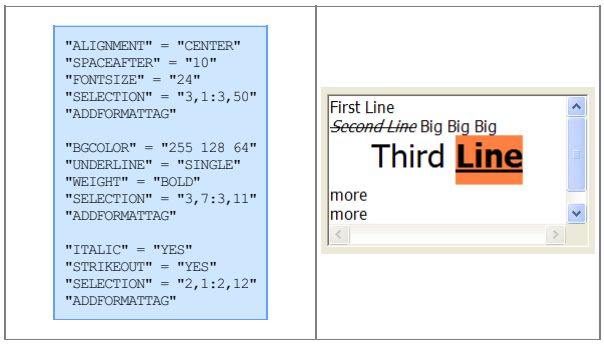
Keys	Action
Navigation	
Arrows	move by individual characters/lines
Ctrl+Arrows	move by words/paragraphs
Home/End	move to begin/end line
Ctrl+Home/End	move to begin/end text
PgUp/PgDn	move vertically by pages
Ctrl+PgUp/PgDn	move horizontally by pages
Selection	
Shift+Arrows	select charaters
Ctrl+A	select all
Deleting	
Del	delete the character at right
Backspace	delete the character at left
Clipboard	
Ctrl+C	copy
Ctrl+X	cut
Ctrl+V	paste

Examples

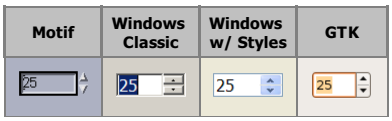
Browse for Example Files



When FORMATTING=YES in Windows or GTK (formatting attributes are set to a formatag object that it is a **IupUser**):



When SPIN=YES:



See Also

IupList, IupMultiLine

FORMATTING [Windows and GTK Only] (non inheritable) (since 3.0)

When enabled allow the use of text formatting attributes. In GTK is always enabled, but only when MULTILINE=YES. Default: NO.

Value

Can be: YES or NO. Default: NO.

Affects

IupText

Auxiliary Attributes

ADDFORMATTAG [write only] (non inheritable)

Name of a format tag element to be added to the **IupText**. The name is associated in C using IupSetHandle. The name association must be done before setting the attribute. It will set the ADDFORMATTAG_HANDLE with the associated handle.

ADDFORMATTAG_HANDLE [write only] (non inheritable)

Handle of a format tag element to be added to the **IupText**. The tag element will be automatically destroyed when the **IupText** is mapped. If the **IupText** is already mapped, the format tag is

immediately destroyed when the attribute is set. The format tag can NOT be reused.

REMOVEFORMATTING [write only] (non inheritable)

Removes the formatting of the current selection if Yes or NULL, and from all text if ALL is used.

Format Tag

The format tag element is a simple IupUser element with some known attributes that will be interpreted when the tag is updated in the native system.

The formatting depends on the existing text, so if VALUE attribute is set, all formatting is lost. You must set it again for the new text.

If the FONT attribute of the **IupText** is set then it will affect the format of all characters in the text.

The default values can not be dynamically changed.

General Format Tag Attributes

BULK: flag that means this tag is composed by several tags as its children. Used to optimize format tag modifications. Default: NO. (since 3.3)

CLEANOUT: when BULK=Yes is used to clear all the formatting at start. Default: NO. (since 3.3)

SELECTION/SELECTIONPOS: same as the **IupText** SELECTION/SELECTIONPOS attributes. If not defined the **IupText** attribute will be used. If the **IupText** attribute is also not defined then the current position will receive the format, so new text inserted or typed will be formatted with the tag (this is not working in GTK). Different tags that use the same selection interval are combined. Setting these attributes here will not change the current setting in **IupText** (since 3.3).

UNITS [Windows Only]: By default all distance units are integers in pixels, but in Windows you can also specify integer units in TWIPs (one twip is 1/1440 of an inch). Can be TWIP or PIXELS. Default: PIXELS.

Paragraph Format Tag Attributes

ALIGNMENT: Can be JUSTIFY, RIGHT, CENTER and LEFT. Default: LEFT.

INDENT: paragraph indentation, the distance between the margin and the paragraph. In Windows the right indentation, and the indentation of the second and subsequent lines (relative to the indentation of the first line) can be independently set using the **INDENTRIGHT** and **INDENTOFFSET** attributes, but only when **INDENT** is set.

LINESPACING: the distance between lines of the same paragraph. In Windows, the values SINGLE, ONEHALF and DOUBLE can be used.

NUMBERING [Windows Only]: Can be BULLET (bullet symbol), ARABIC (arabic numbers - 1,2,3...), LCLETTER (lowercase letters - a,b,c...), UCLETTER (uppercase letters - A,B,C...), LCROMAN (lowercase Roman numerals - i,ii,iii...), UCROMAN (uppercase Roman numerals - I,II,III...) and NONE. Default: NONE.

NUMBERINGSTYLE [Windows Only]: Can be RIGHTPARENTESES "a)", PARENTESES "(a)", PERIOD "a.", NONUMBER (it will skip the numbering or bullet for the item) and NONE "". Default: NONE.

NUMBERINGTAB [Windows Only]: Minimum distance from a paragraph numbering or bullet to the paragraph text.

SPACEAFTER: distance left empty above the paragraph.

SPACEBEFORE: distance left empty below the paragraph.

TABSARRAY: a sequence of tab positions and alignment up to 32 tabs. It uses the format:"pos align pos align pos align...". Position is the distance relative to the left margin and alignment can be LEFT, CENTER, RIGHT and DECIMAL. In GTK only LEFT is currently supported. When DECIMAL alignment is used, the text is aligned according to a decimal point or period in the text, it is normally used to align numbers.

Character Format Tag Attributes

BGCOLOR: string containing a color in the format "rrr ggg bbb" for the background of the text.

DISABLED [Windows Only]: Can be YES or NO. Default NO. Set the visual appearance to disabled.

FGCOLOR: string containing a color in the format "rrr ggg bbb" for the text.

FONTSCALE: a size scale relative to the selected or current size. Values greatter than 1 will increase the font. Values smaller than 1 will shirnk the font. Default: 1.0. The following values are also accpeted: "XX-SMALL" (0.58), "X-SMALL" (0.64), "SMALL" (0.83), "MEDIUM" (1.0), "LARGE" (1.2), "X-LARGE" (1.44), "XX-LARGE" (1.73).

FONTFACE: the face name of the font.

FONTSIZE: the size of the font in pixels or points. Pixel size uses negative values.

ITALIC: Can be YES or NO. Default NO.

LANGUAGE [GTK Only]: A text with a description of the text language. The same value can be used in the "SYSTEMLANGUAGE" global attribute.

RISE: the distance, positive or negative from the base line. Can also use the values SUPERSCRIPT and SUBSCRIPT, but this values will also reduce the size of the font.

SMALLCAPS [GTK Only]: Can be YES or NO. Default NO. (Does not work always, depends on the font)

PROTECTED: Can be YES or NO. Default NO. When set to YES the selected text can NOT be edited.

STRETCH [GTK Only]: Can be EXTRA_CONDENSED, CONDENSED, SEMI_CONDENSED, NORMAL, SEMI_EXPANDED, EXPANDED and EXTRA_EXPANDED. Default NORMAL. (Does not work always, depends on the font)

STRIKEOUT: Can be YES or NO. Default NO.

UNDERLINE: Can be SINGLE, DOUBLE, DOTTED or NONE. Default NONE. DOTTED is supported only in Windows.

WEIGHT: Can be EXTRALIGHT, LIGHT, NORMAL, SEMIBOLD, BOLD, EXTRABOLD and HEAVY. Default: NORMAL.

Examples

In C:

```
Ihandle* formattag;
IupSetAttribute(text, "FORMATTING", "YES");

formattag = IupUser();
IupSetAttribute(formattag, "ALIGNMENT", "CENTER");
IupSetAttribute(formattag, "SPACEAFTER", "10");
IupSetAttribute(formattag, "FONTSIZE", "24");
IupSetAttribute(formattag, "SELECTION", "3,1:3,50");
IupSetAttribute(text, "ADDFORMATTAG_HANDLE", (char*)formattag);

formattag = IupUser();
IupSetAttribute(formattag, "BGCOLOR", "255 128 64");
IupSetAttribute(formattag, "UNDERLINE", "SINGLE");
```

```
IupSetAttribute(formattag, "WEIGHT", "BOLD");
IupSetAttribute(formattag, "SELECTION", "3,7:3,11");
IupSetAttribute(text, "ADDFORMATTAG_HANDLE", (char*)formattag);

formattag = IupUser();
IupSetAttribute(formattag, "ITALIC", "YES");
IupSetAttribute(formattag, "STRIKEOUT", "YES");
IupSetAttribute(formattag, "SELECTION", "2,1:2,12");
IupSetAttribute(text, "ADDFORMATTAG_HANDLE", (char*)formattag);
```

In Lua using BULK:

```
tags = iup.user { bulk = "Yes", cleanout = "Yes" }
iup.Append(tags, iup.user { selectionpos = "0:3", fgcolor = "255 0 0"})
iup.Append(tags, iup.user { selectionpos = "5:10", fgcolor = "0 0 255"})
text.addformattag = tags
```

Check the Indentation library created by Kristofer Karlsson and ported to IUP by Nicolas Noble that adds syntax highlighting to a Lua code text in a **IupText** control. It is not fast because it process the entire text from time to time. For example:

```
require"indent" -- indent.lua must be available
text = iup.text { multiline = "Yes", font = "Courier", expand = "Yes", value = someluacode }
IndentationLib.enable(text)
```

MASK (non inheritable) (since 3.0)

Defines a mask that will filter interactive text input.

Value

string

Set to NULL to remove the mask.

Notes

Since the validation process is performed key by key when the user is typing, an intermediate value cannot be typed if it does not follow the mask rules.

If you set the VALUE attribute any text can be used. To set a value that is validated by the current MASK use VALUEMASKED.

Pre-Defined Masks

Definition	Value	Description
IUP_MASK_INT	"[+/-]?/d+"	integer number
IUP_MASK_UINT	"/d+"	unsigned integer number
IUP_MASK_FLOAT	"[+/-]?(/d+/.?/d*|/./d+)"	floating point number
IUP_MASK_UFLOAT	"(/d+/.?/d*|/./d+)"	unsigned floating point number
IUP_MASK_EFLOAT	"[+/-]?(/d+/.?/d*|/./d+)([eE][+/-]?/d+)?"	floating point number with exponential notation
IUP_MASK_FLOATCOMMA	"[+/-]?(/d+/,?/d*|/,/d+)"	floating point number
IUP_MASK_UFLOATCOMMA	"(/d+/,?/d*|/,/d+)"	unsigned floating point number

Auxiliary Attributes

MASKCASEI (non inheritable)

If YES, will turn the filter case insensitive. Default: NO.

MASKNOEMPTY (non inheritable) (since 3.17)

If YES, value can NOT be NULL or empty. Default: NO (can be empty or NULL).

MASKDECIMALSYMBOL (non inheritable) (since 3.13)

The decimal symbol for string/float conversion. Can be "." or ",". Must be set before MASKFLOAT.

MASKINT (non inheritable) (write only)

Defines an integer mask with limits. Format: "%d:%d" ("min:max"). It will replace MASK using one of the pre-defined masks.

MASKFLOAT (non inheritable) (write only)

Defines a floating point mask with limits. Format: "%g:%g" ("min:max"). It will replace MASK using one of the pre-defined masks.

Auxiliary Callbacks

MASKFAIL_CB: Action generated when the new text fails at the mask check. (since 3.9)

```
int function(Ihandle *ih, char *new_value); [in C]
elem:maskfail(new_value: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
new_value: Represents the new text value.

Pattern Specification

The pattern to be searched in the text can be defined by the rules given below.

- "Function" codes (such as /l, /D, /w) cannot be used inside a class ([...]).
- If the character following a / does not mean a special case (such as /w or /n), it is matched with no / - that means that /x will match only x, and not /x. If you want to match /x, use //x.
- The caret (^) character has different meanings when used inside or outside a class - inside a class it means negative, and outside a class it is an anchor to the beginning of a line.

- The boundary function (/b) anchors the pattern to a word boundary - it does not match anything. A word boundary is a point between a /w and a /W character.
- Capture operators (f and g) group patterns and are also used to keep matched sections of texts.
- A word on precedence: concatenation has precedence over the alternation (j) operator - that is, faj fej fi will match fa OR fe OR fi.
- The @ character is used to determine that, instead of searching the text until the first match is made, the function should try to match the pattern only with the first character. If present, it must be the first character of the pattern.
- The % character is used to determine that the text should be searched to its end, independently of the number of matches found. If present, it must be the first character of the pattern. This is only useful when combined with the capture feature.

Allowed pattern characters

c	Matches a "c" (non-special) character
.	Matches any single character
[abc]	Matches an "a", "b" or "c" characters
[a-d]	Matches any character between "a" and "d", including them (just like [abcd])
[^a-dg]	Matches any character which is neither between "a" and "d" nor "a" "g"
/d	Matches any digit (just like [0-9])
/D	Matches any non-digit (just like [^0-9])
/l	Matches any letter (just like [a-zA-Z])
/L	Matches any non-letter (just like [^a-zA-Z])
/w	Matches any alphanumeric character (just like [0-9a-zA-Z])
/W	Matches any non-alphanumeric character (just like [^0-9a-zA-Z])
/s	Matches any "blank" character (TAB, SPACE, CR)
/S	Matches any non-blank character
/n	Matches a newline character
/t	Matches a tabulation character
/nnn	Matches an ASCII character with a nnn value (decimal)
/xnn	Matches an ASCII character with a nn value (hexadecimal)
/special	Matches the special character literally (/[, //, /.)
abc	Matches a sequence of a, b and c patterns in order
aj bj c	Matches a pattern a, b or c
a*	Matches 0 or more characters a
a+	Matches 1 or more characters a
a?	Matches 1 or no characters a
(pattern)	Considers pattern as one character for the above
fpatterng	Captures pattern for later reference
/b	Anchors to a word boundary
/B	Anchors to a non-boundary
^pattern	Anchors pattern to the beginning of a line
pattern$	Anchors pattern to the end of a line
@pattern	Returns the match found only in the beginning of the text
%pattern	Returns the first match found, but searches all the text

Examples

(my|his)	Matches both my pattern and his pattern.
/d/d:/d/d(:/d/d)?	Matches time with seconds (01:25:32) or without seconds (02:30).
[A-D]/l+	Matches names such as Australia, Bolivia, Canada or Denmark, but not England, Spain or single letters such as A.
/l/w*	my variable = 23 * width;
^Subject:[^/n]*/n	Subject: How to match a subject line.1
/b[ABab]/w*	Matches any word that begins with A or B
from:/s*/w+	Captures "sender" in a message from sender

Affects

IupText, IupMultiline, IupList and IupMatrix

IupMultiLine (same as IupText with MULTILINE=YES since IUP 3.0)

Creates an editable field with one or more lines.

Since IUP 3.0, **IupText** has support for multiple lines when the MULTILINE attribute is set to YES. Now when a **IupMultiline** element is created in fact a **IupText** element with MULTILINE=YES is created.

See IupText

Creation

```
Ihandle* IupMultiLine(const char *action); [in C]
iup.multiline{} -> (ih: ihandle) [in Lua]
multiline(action) [in LED]
```

action: name of the action generated when the user types something. It can be NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

Examples

Browse for Example Files

IupToggle

Creates the toggle interface element. It is a two-state (on/off) button that, when selected, generates an action that activates a function in the associated application. Its visual representation can contain a text or an image.

Creation

```
Ihandle* IupToggle(const char *title, const char *action); [in C]
iup.toggle{[title = title: string]} -> (ih: ihandle) [in Lua]
toggle(title, action) [in LED]
```

title: Text to be shown on the toggle. It can be NULL. It will set the TITLE attribute.
action: name of the action generated when the toggle is selected. It can be NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

ALIGNMENT (non inheritable): horizontal and vertical alignment when IMAGE is defined. Possible values: "ALEFT", "ACENTER" and "ARIGHT", combined to "ATOP", "ACENTER" and "ABOTTOM". Default: "ACENTER:ACENTER". Partial values are also accepted, like "ARIGHT" or ":ATOP", the other value will be used from the current alignment. In Motif, vertical alignment is restricted to "ACENTER". In Windows works only when Visual Styles is active. Text is always left aligned. (since 3.0)

BGCOLOR: Background color of toggle mark when displaying a text. The text background is transparent, it will use the background color of the native parent. When displaying an image in Windows the background is ignored and the system color is used. Default: the global attribute DLGBGCOLOR.

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. In Windows the control will still get the focus when clicked. Default: YES. (since 3.0)

FGCOLOR: Color of the text shown on the toggle. In Windows, when using Visual Styles FGCOLOR is ignored. Default: the global attribute DLGFGCOLOR.

FLAT (creation only): Hides the toggle borders until the mouse enter the toggle area when the toggle is not checked. If the toggle is checked, then the borders will be shown even if flat is enabled. Used only when IMAGE is defined. Can be YES or NO. Default: NO. (since 3.3)

IMAGE (non inheritable): Image name. When the IMAGE attribute is defined, the TITLE is not shown. This makes the toggle looks just like a button with an image, but its behavior remains the same. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. (GTK 2.6)

IMPRESS (non inheritable): Image name of the pressed toggle. Unlike buttons, toggles always display the button border when IMAGE and IMPRESS are both defined. (GTK 2.6)

IMINACTIVE (non inheritable): Image name of the inactive toggle. If it is not defined but IMAGE is defined then for inactive toggles the colors will be replaced by a modified version of the background color creating the disabled effect. (GTK 2.6)

MARKUP [GTK only]: allows the title string to contains pango markup commands. Works only if a mnemonic is NOT defined in the title. Can be "YES" or "NO". Default: "NO".

PADDING: internal margin when IMAGE is defined. Works just like the MARGIN attribute of the **IupHbox** and **IupVbox** containers, but uses a different name to avoid inheritance problems. Default value: "0x0". (since 3.0)

RADIO (read-only): returns if the toggle is inside a radio. Can be "YES" or "NO". Valid only after the element is mapped, before returns NULL. (since 3.0)

RIGHTBUTTON (Windows Only) (creation only): place the check button at the right of the text. Can be "YES" or "NO". Default: "NO".

VALUE (non inheritable): Toggle's state. Values can be "ON", "OFF" or "TOGGLE". If 3STATE=YES then can also be "NOTDEF". Default: "OFF". The TOGGLE option will invert the current state (since 3.7). In GTK if you change the state of a radio, the unchecked toggle will receive an ACTION callback notification. Can only be set to Yes for a toggle inside a radio, it will automatically set to OFF the previous toggle that was ON.

TITLE (non inheritable): Toggle's text. If IMAGE is not defined before map, then the default behavior is to contain a text. The button behavior can not be changed after map. The natural size will be larger enough to include all the text in the selected font, even using multiple lines, plus the button borders or check box if any. The '\n' character is accepted for line change. The "&" character can be used to define a mnemonic, the next character will be used as key. Use "&&" to show the "&" character instead on defining a mnemonic. The toggle can be activated from any control in the dialog using the "Alt+key" combination. (mnemonic support since 3.0)

3STATE (creation only): Enable a three state toggle. Valid for toggles with text only and that do not belong to a radio. Can be "YES" or NO". Default: "NO".

ACTIVE, FONT, EXPAND, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

ACTION: Action generated when the toggle's state (on/off) was changed. The callback also receives the toggle's state.

```
int function(Ihandle* ih, int state); [in C]
ih:action(state: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
state: 1 if the toggle's state was shifted to on; 0 if it was shifted to off.

Returns: IUP_CLOSE will be processed.

VALUECHANGED_CB: Called after the value was interactively changed by the user. Called after the ACTION callback, but under the same context. (since 3.0)

```
int function(Ihandle *ih); [in C]
ih:valuechanged_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

Toggle with image or text can not change its behavior after mapped. This is a creation attribute. But after creation the image can be changed for another image, and the text for another text.

Toggles are activated using the Space key.

To build a set of mutual exclusive toggles, insert them in an **IupRadio** container. They must be inserted before creation, and their behavior can not be changed. If you need to dynamically remove toggles that belongs to a radio in Windows, then put the radio inside an **IupFrame** that has a title.

A toggle that is a child of an **IupRadio** automatically receives a name when its is mapped into the native system. (since 3.16)

Examples

Browse for Example Files

	Windows	Windows	

See Also

IupImage, IupButton, IupLabel, IupRadio.

IupTree Attributes

General

ADDEXPANDED (non inheritable): Defines if branches will be expanded when created. The branch will be actually expanded when it receives the first child. Possible values: "YES" = The branches will be created expanded; "NO" = The branches will be created collapsed. Default: "YES".

ADDROOT (non inheritable): automatically adds an empty branch as the first node when the tree is mapped. Default: "YES". (Since 3.1)

AUTOREDRAW [Windows] (non inheritable): automatically redraws the tree when something has change. Set to NO to add many items to the tree without updating the display. Default: "YES". (since 3.3)

BGCOLOR: Background color of the tree. Default: the global attribute TXTBGCOLOR.

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. In Windows the control will still get the focus when clicked. Default: YES. (since 3.0)

COUNT (read only) (non inheritable): returns the total number of nodes in the tree. (since 3.0)

DRAGDROPTREE (non inheritable): enable or disable the drag and drop of nodes between trees, in the same IUP application. Drag & Drop attributes must be set in order to activate the drag & drop support. On the other hand, it is not necessary to register drag & drop callbacks. Default: NO. (since 3.10)

DROPFILESTARGET [Windows and GTK Only] (non inheritable): Enable or disable the drop of files. Default: NO, but if DROPFILES_CB is defined when the element is mapped then it will be automatically enabled. This is NOT related to the drag&drop of nodes inside the tree. (since 3.0)

DROPEQUALDRAG (non inheritable): if enabled will allow a drop node to be equal to the drag node. Used only if SHOWDRAGDROP =Yes. In the case the nodes are equal the callback return value is ignored and nothing is done after. (since 3.3)

EMPTYAS3STATE (non inheritable) [Windows Only]: when SHOWTOGGLE=Yes, the empty space left in nodes that NODEVISIBLEid=NO is filled with the image of the 3state toggle. Can be Yes or NO. Default: No. (since 3.11.2)

EXPAND (non inheritable): The default value is "YES".

FGCOLOR: default text foreground color. Once each node is created it will not change its color when FGCOLOR is changed. Default: the global attribute TXTFGCOLOR. (since 3.0)

HIDEBUTTONS (creation only): hide the expand and collapse buttons. In GTK, branches will be only expanded programmatically. In Motif it did not work and crash the test. (since 3.0) (GTK 2.12)

HIDELINES (creation only): hide the lines that connect the nodes in the hierarchy. (since 3.0) (GTK 2.10)

HLCOLOR [Windows and Motif Only] (non inheritable): the background color of the selected node. Default: TXTHLCOLOR global attribute. (since 3.16)

INDENTATION: sets the indentation level in pixels. The visual effect of changing the indentation is highly system dependent. In GTK it acts as an additional indent value, and the lines do not follow the extra indent. In Windows is limited to a minimum of 5 pixels. (since 3.0) (GTK 2.12)

INFOTIP [Windows Only]: the TIP is shown every time an item is highlighted. This is the default behavior for TIPs in native tree controls in Windows, if set to No then it will use the regular TIP behavior. Default: Yes. (since 3.14)

RASTERSIZE (non inheritable): the initial size is "400x200". Set to NULL to allow the automatic layout use smaller values.

SHOWDRAGDROP (creation only) (non inheritable): Enables the internal drag and drop of nodes, and enables the **DRAGDROP_CB** callback. Default: "NO". Works only if MARKMODE=SINGLE.

SHOWTOGGLE (creation only) (non inheritable): enables the use of toggles for all nodes of the tree. Can be "YES", "3STATE" or NO". Default: "NO". In Motif Versions 2.1.x and 2.2.x, the images are disabled (toggle and text only are drawn in nodes of the tree). (since 3.6)

SPACING: vertical internal padding for each node. Notice that the distance between each node will be actually 2x the spacing. (since 3.0)

TOPITEM (write-only): position the given node identifier at the top of the tree or near to make it visible. If any parent node is collapsed then they are automatically expanded. (since 3.0)

ACTIVE, EXPAND, FONT, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

The NAME common attribute is still not supported because of a backward compatibility code. Old applications must change the use of the old NAME attribute to TITLE, so the new NAME common attribute can be enabled in future versions.

Drag & Drop attributes are supported, but SHOWDRAGDROP must be set no No.

Nodes (non inheritable)

For these attributes "id" is the specified node identifier. If "id" is empty or invalid, then the focus node is used as the specified node.

CHILDCOUNTid (read only): returns the immediate children count of the specified branch. It does not count children of child that are branches. (since 3.0)

COLORid: text foreground color of the specified node. The value should be a string in the format "R G B" where R, G, B are numbers from 0 to 255.

DEPTHid (read only): returns the depth of the specified node. The first node has depth=0, its immediate children has depth=1, their children has depth=2 and so on.

KINDid (read only): returns the kind of the specified node. Possible values:

- "LEAF": The node is a leaf
- "BRANCH": The node is a branch

PARENTid (read only): returns the identifier of the specified node.

STATEid: the state of the specified branch. Returns NULL for a LEAF. In Windows, it will be effective only if the branch has children. In GTK, it will be effective only if the parent is expanded. Possible values:

- "EXPANDED": Expanded branch state (shows its children)
- "COLLAPSED": Collapsed branch state (hides its children)

TITLEid: the text label of the specified node.

TITLEFONTid: the text font of the specified node. The format is the same as the FONT attribute. (since 3.0)

TOGGLEVALUEid (non inheritable): defines the toggle state. Values can be "ON" or "OFF". If SHOW3STATE=YES then can also be "NOTDEF". Default: "OFF". (Since 3.6)

TOGGLEVISIBLEid (non inheritable): defines the toggle visible state. Values can be "Yes" or "No". Default: "Yes". (Since 3.8)

TOTALCHILDCOUNTid (read only): returns the total children count of the specified branch. It counts all grandchildren. (since 3.0)

USERDATAid: the user data associated with the specified node. (since 3.0)

Images (non inheritable)

IMAGEid (write only): image name to be used in the specified node, where id is the specified node identifier. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. In Windows and Motif set the BGCOLOR attribute before setting the image. If node is a branch it is used when collapsed.

IMAGEEXPANDEDid (write only): same as the IMAGE attribute but used for expanded branches.

IMAGELEAF: the image name that will be shown for all leaves. Default: "IMGLEAF". Internal values "IMGBLANK" and "IMGPAPER" are also available. If BGCOLOR is set the image is automatically updated.

IMAGEBRANCHCOLLAPSED: the image name that will be shown for all collapsed branches. Default: "IMGCOLLAPSED". If BGCOLOR is set the image is automatically updated.

IMAGEBRANCHEXPANDED: the image name that will be shown for all expanded branches. Default: "IMGEXPANDED". If BGCOLOR is set the image is automatically updated.

Focus Node

VALUE (non inheritable): The focus node identifier. When retrieved but there isn't a node with focus it returns 0 if there are any nodes, and returns -1 if there are no nodes. When changed and MARKMODE=SINGLE the node is also selected. The tree is always scrolled so the node becomes visible. In Motif the tree will also receive the focus. Additionally accepts the values:

"ROOT" or "FIRST": the first node
"LAST": the last visible node
"NEXT": the next visible node, one node after the focus node. If at the last does nothing
"PREVIOUS": the previous visible node, one node before the focus node. If at the first does nothing
"PGDN": the next visible node, ten nodes node after the focus node. If at the last does nothing
"PGUP": the previous visible node, ten nodes before the focus node. If at the first does nothing

Marks

MARK (write only) (non inheritable): Selects a range of nodes in the format "start-end" (%d-%d). Allowed only when MARKMODE=MULTIPLE. Also accepts the values:

"INVERTid": Inverts the specified node selected state, where id is the specified node identifier. If id is empty or invalid, then the focus node is used as reference node.
"BLOCK": Selects all nodes between the focus node and the initial block-marking node defined by MARKSTART
"CLEARALL": Clear the selection of all nodes
"MARKALL": Selects all nodes
"INVERTALL": Inverts the selection of all nodes

MARKEDid (non inheritable): The selection state of the specified node, where id is the specified node identifier. If id is empty or invalid, then the focus node is used as reference node. Can be: YES or NO. Default: NO

MARKEDNODES (non inheritable): The selection state of all nodes when MARKMODE=MULTIPLE. It is/accepts a sequence of '+' and '-' symbols indicating the state of each item ('+'=selected, '-'=unselected. When setting this value, if the number of specified symbols is smaller than the total count then the remaining nodes will not be changed. (since 3.1)

MARKMODE: defines how the nodes can be selected. Can be: SINGLE or MULTIPLE. Default: SINGLE.

MARKSTART (non inheritable): Defines the initial node for the block marking, used when MARK=BLOCK. The value must be the node identifier. Default: 0 (first node).

MARKWHENTOGGLE (non inheritable) [GTK and Windows Only]: selects or clears the selection of a node when its toggle is changed. Works only if the node has a toggle. Default: No. (Since 3.17)

Hierarchy (non inheritable)

For these attributes "id" is the specified node identifier. If "id" is empty or invalid, then the focus node is used as the specified node.

ADDLEAFid (write only): Adds a new leaf after the reference node, where id is the reference node identifier. Use id=-1 to add before the first node. The value is used as the text label of the new node. The id of the new node will be the id of the reference node + 1. The attribute **LASTADDNODE** is set to the new id. The reference node is marked and all others unmarked. The reference node position remains the same. If the reference node does not exist, nothing happens. If the reference node is a branch then the depth of the new node is one depth increment from the depth of the reference node, if the reference node is a leaf then the new node has the same depth. If you need to add a node after a specified node but at a different depth use **INSERTLEAF**. Ignored if set before map.

ADDBRANCHid (write only): Same as **ADDLEAF** for branches. Branches can be created expanded or collapsed depending on **ADDEXPANDED**. Ignored if set before map.

COPYNODEid (write only): Copies a node and its children, where id is the specified node identifier. The value is the destination node identifier. If the destination node is a branch and it is expanded, then the specified node is inserted as the first child of the destination node. If the branch is not expanded or the destination node is a leaf, then it is inserted as the next brother of the leaf. The specified node is not changed. All node attributes are copied, except user data. Ignored if set before map. (since 3.0)

DELNODEid (write only): Removes a node and/or its children, where id is the specified node identifier. Ignored if set before map. Possible values:

- "ALL": deletes all nodes, id is ignored (Since 3.1)
- "SELECTED": deletes the specified node and its children
- "CHILDREN": deletes only the children of the specified node
- "MARKED": deletes all the selected nodes (and all their children), id is ignored

EXPANDALL (write only): expand or contracts all nodes. Can be YES (expand all), or NO (contract all). (since 3.0)

INSERTLEAFid, **INSERTBRANCHid** (write only): Same as **ADDLEAF** and **ADDBRANCH** but the depth of the new node is always the same of the reference node. If the reference node is a leaf, then the id of the new node will be the id of the reference node + 1. If the reference node is a branch the id of the new node will be the id of the reference node + 1 + the total number of child nodes of the reference node. (since 3.0)

MOVENODEid (write only): Moves a node and its children, where id is the specified node identifier. The value is the destination node identifier. If the destination node is a branch and it is expanded, then the specified node is inserted as the first child of the destination node. If the branch is not expanded or the destination node is a leaf, then it is inserted as the next brother of the leaf. The specified node is removed. User data and all node attributes are preserved. Ignored if set before map. (since 3.0)

Editing

RENAME (write only): Forces a rename action to take place. Valid only when SHOWRENAME=YES.

RENAMECARET (write only): the caret's position of the text box when in-place renaming. Same as the CARET attribute for IupText, but here is used only once after SHOWRENAME_CB is called and before the text box is shown.

RENAMESELECTION (write only): the selection interval of the text box when in-place renaming. Same as the SELECTION attribute for IupText, but here is used only once after SHOWRENAME_CB is called and before the text box is shown.

SHOWRENAME (creation in Windows) (non inheritable): Allows the in place rename of a node. Default: "NO". Since IUP 3.0, F2 and clicking twice only starts to rename a node if SHOWRENAME=Yes. In Windows must be set to YES before map, but can be changed later (since 3.3).

IupTree Callbacks

SELECTION_CB: Action generated when a node is selected or deselected. This action occurs when the user clicks with the mouse or uses the keyboard with the appropriate combination of keys. It may be called more than once for the same node with the same status.

```
int function(Ihandle *ih, int id, int status) [in C]
ih:selection_cb(id, status: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
id: Node identifier.
status: 1=node selected, 0=node unselected.

MULTISELECTION_CB: Action generated after a continuous range of nodes is selected in one single operation. If not defined the SELECTION_CB with status=1 will be called for all nodes in the range. The range is always completely included, independent if some nodes were already marked. That single operation also guaranties that all other nodes outside the range are already not selected. Called only if MARKMODE=MULTIPLE.

```
int function(Ihandle *ih, int* ids, int n) [in C]
ih:multiselection_cb(ids: table, n: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
ids: Array of node identifiers. This array is kept for backward compatibility, the range is simply defined by ids[0] to ids[n-1], where `ids[i+1]=ids[i]+1`.
n: Number of nodes in the array.

MULTIUNSELECTION_CB: Action generated before multiple nodes are unselected in one single operation. If not defined the SELECTION_CB with status=0 will be called for all nodes in the range. The range is not necessarily continuous. Called only if MARKMODE=MULTIPLE. (Since 3.1)

```
int function(Ihandle *ih, int* ids, int n) [in C]
ih:multiunselection_cb(ids: table, n: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
ids: Array of node identifiers.
n: Number of nodes in the array.

BRANCHOPEN_CB: Action generated when a branch is expanded. This action occurs when the user clicks the "+" sign on the left of the branch, or when double clicks the branch, or hits Enter on a collapsed branch.

```
int function(Ihandle *ih, int id) [in C]
ih:branchopen_cb(id: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
id: node identifier.

Returns: IUP_IGNORE for the branch not to be opened, or IUP_DEFAULT for the branch to be opened.

BRANCHCLOSE_CB: Action generated when a branch is collapsed. This action occurs when the user clicks the "-" sign on the left of the branch, or when double clicks the branch, or hits Enter on an expanded branch.

```
int function(Ihandle *ih, int id); [in C]
ih:branchclose_cb(id: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
id: node identifier.

Returns: IUP_IGNORE for the branch not to be closed, or IUP_DEFAULT for the branch to be closed.

EXECUTELEAF_CB: Action generated when a leaf is to be executed. This action occurs when the user double clicks a leaf, or hits Enter on a leaf.

```
int function(Ihandle *ih, int id); [in C]
ih:executeleaf_cb(id: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
id: node identifier.

SHOWRENAME_CB: Action generated when a node is about to be renamed. It occurs when the user clicks twice the node or press **F2**. Called only if SHOWRENAME=YES.

```
int function(Ihandle *ih, int id); [in C]
elem:showrename_cb(id: number: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
id: node identifier.

Returns: if IUP_IGNORE is returned, the rename is canceled (in GTK the rename continuous but the edit box is read-only).

RENAME_CB: Action generated after a node was renamed in place. It occurs when the user press **Enter** after editing the name, or when the text box looses it focus. Called only if SHOWRENAME=YES.

```
int function(Ihandle *ih, int id, char *title); [in C]
elem:rename_cb(id: number, title: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
id: node identifier.
title: new node title.

Returns: The new title is accepted only if the callback returns IUP_DEFAULT. If the callback does not exists the new title is always accepted. If the user pressed **Enter** and the callback returns IUP_IGNORE the editing continues. If the text box looses its focus the editing stops always.

DRAGDROP_CB: Action generated when an internal drag & drop is executed. Only active if **SHOWDRAGDROP=YES.**

```
int function(Ihandle *ih, int drag_id, int drop_id, int isshift, int iscontrol); [in C]
ih:dragdrop_cb(drag_id, drop_id, isshift, iscontrol: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

drag_id: Identifier of the clicked node where the drag start.
drop_id: Identifier of the clicked node where the drop were executed. -1 indicates a drop in a blank area.
isshift: flag indicating the shift key state.
iscontrol: flag indicating the control key state.

Returns: if returns IUP_CONTINUE, or if the callback is not defined and **SHOWDRAGDROP=YES**, then the node is moved to the new position. If Ctrl is pressed then the node is copied instead of moved. If the drop node is a branch and it is expanded, then the drag node is inserted as the first child of the node. If the branch is not expanded or the node is a leaf, then the drag node is inserted as the next brother of the drop node.

NODEREMOVED_CB: Action generated when a node is going to be removed. It is only a notification, the action can not be aborted. No node dependent attribute can be consulted during the callback. Not called when the tree is unmapped. It is useful to remove memory allocated for the userdata. (since 3.0)

```
int function(Ihandle *ih, void* userdata); [in C]
ih:noderemoved_cb(userid: userdata/table) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
userdata/userid: USERDATA attribute in C, or userid object in Lua.

RIGHTCLICK_CB: Action generated when the right mouse button is pressed over a node.

```
int function(Ihandle *ih, int id); [in C]
ih:rightclick_cb(id: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
id: node identifier.

TOGGLEVALUE_CB: Action generated when the toggle's state was changed. The callback also receives the new toggle's state. (since 3.6)

```
int function(Ihandle *ih, int id, int state); [in C]
elem:togglevalue_cb(id, state: number: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
id: node identifier.
state: 1 if the toggle's state was shifted to ON; 0 if it was shifted to OFF. If SHOW3STATE=YES, −1 if it was shifted to NOTDEF.

BUTTON_CB: Action generated when any mouse button is pressed or released inside the element. Use IupConvertXYToPos to convert (x,y) coordinates in the node identifier. (since 3.0)

MOTION_CB: Action generated when the mouse is moved over the element. Use IupConvertXYToPos to convert (x,y) coordinates in item the node identifier. (since 3.0)

DROPFILES_CB [Windows and GTK Only]: Action generated when one or more files are dropped in the element. (since 3.0)

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

In Motif the tree always resets the focus to the first node when receive the focus. The KILLFOCUS_CB callback is called only when the focus is at the first node. Also in Motif some LEAVEWINDOW_CB events are delayed to when the user enter again, firing a leave and enter events at enter time.

Drag & Drop callbacks are supported, but SHOWDRAGDROP must be set to NO.

orientation: optional orientation of valuator. Can be NULL. See ORIENTATION attribute.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BGCOLOR: transparent in all systems except in Motif. It will use the background color of the native parent.

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. In Windows the control will still get the focus when clicked. Default: YES. (since 3.0)

INVERTED: Invert the minimum and maximum positions on screen. When INVERTED=YES maximum is at top and left (minimum is bottom and right), when INVERTED=NO maximum is at bottom and right (minimum is top and left). The initial value depends on ORIENTATION passed as parameter on creation, if ORIENTATION=VERTICAL default is YES, if ORIENTATION=HORIZONTAL default is NO. (since 3.0)

MAX: Contains the maximum valuator value. Default is "1". When changed the display will not be updated until VALUE is set.

MIN: Contains the minimum valuator value. Default is "0". When changed the display will not be updated until VALUE is set.

PAGESTEP: Controls the increment for pagedown and pageup keys. It is not the size of the increment. The increment size is "pagestep*(max-min)", so it must be 0<pagestep<1. Default is "0.1".

RASTERSIZE (non inheritable): The initial size is 100 pixels along the major axis, and the handler normal size on the minor axis. If there are ticks then they are added to the natural size on the minor axis. The handler can be smaller than the normal size. Set to NULL to allow the automatic layout use smaller values.

SHOWTICKS [Windows and Motif Only]: The number of tick marks along the valuator trail. Minimum value is "2". Default is "0", in this case the ticks are not shown. It can not be changed to 0 from a non zero value, or vice-versa, after the control is mapped. GTK does not support ticks.

STEP: Controls the increment for keyboard control and the mouse wheel. It is not the size of the increment. The increment size is "step*(max-min)", so it must be 0<step<1. Default is "0.01".

TICKSPOS [Windows Only] (creation only): Allows to position the ticks in both sides (BOTH) or in the reverse side (REVERSE). Default: NORMAL. The normal position for horizontal orientation is at the top of the control, and for vertical orientation is at the left of the control. In Motif, the ticks position is always normal. (since 3.0)

ORIENTATION (non inheritable): Informs whether the valuator is "VERTICAL" or "HORIZONTAL". Vertical valuators are bottom to up, and horizontal valuators are left to right variations of min to max (but can be inverted using INVERTED). Default: "HORIZONTAL".

VALUE (non inheritable): Contains a number between MIN and MAX, indicating the valuator position. Default: "0.0".

ACTIVE, EXPAND, FONT, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, ZORDER, VISIBLE: also accepted.

Callbacks

VALUECHANGED_CB: Called after the value was interactively changed by the user.

```
int function(Ihandle *ih); [in C]
ih:valuechanged_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

This control replaces the old IupVal implemented in the additional controls. The old callbacks are still supported but called only if the VALUECHANGED_CB callback is not defined. The MOUSEMOVE_CB callback is only called when the user moves the handler using the mouse. The BUTTON_PRESS_CB callback is called only when the user press a key that changes the position of the handler. The BUTTON_RELEASE_CB callback is called only when the user release the mouse button after moving the handler.

In Motif, after the user clicks the handler a KILLFOCUS will be ignored when the control loses its focus.

Keyboard Mapping

This is the default mapping when INVERTED has the default value, or ORIENTATION=HORIZONTAL+INVERTED=NO.

Keys	Action for HORIZONTAL
Right Arrow	move right, increment by one step
Left Arrow	move left, decrement by one step
Ctrl+Right Arrow or PgDn	move right, increment by one page step
Ctrl+Left Arrow or PgUp	move left, decrement by one page step
Home	move all left, set to minimum
End	move all right, set to maximum

This is the default mapping when INVERTED has the default value, or ORIENTATION=VERTICAL+INVERTED=YES.

Keys	Action for VERTICAL
Up Arrow	move up, increment by one step
Down Arrow	move down, decrement by one step
Ctrl+Up Arrow or PgUp	move up, increment by one page step
Ctrl+Down Arrow or PgDn	move down, decrement by one page step
Home	move all up, set to maximum
End	move all down, set to minimum

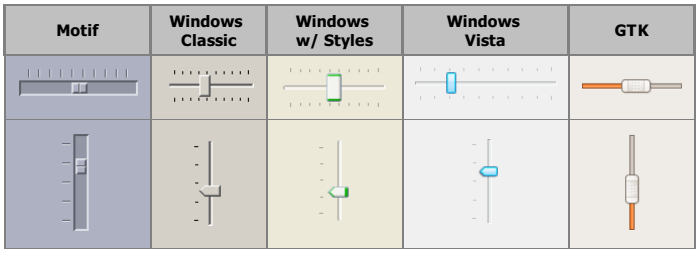
Visually all the keys move to the same direction independent from the INVERTED attribute.

Semantically all the keys change the value depending on the INVERTED attribute.

This behavior is slightly different from the defined by the native systems (Home and End keys are different). But it is the same in all systems.

Examples

Browse for Example Files



IupControls

Controls Library

Several additional controls are included in this library. These controls are drawn by IUP using CD on a IupCanvas control, and are not native controls.

The **iupcontrols.h** file must be included in the source code. If you plan to use the control in Lua, you should also include **iupluacontrols.h**.

The **IupControlsOpen** function must be called after **IupOpen**. To make the controls available in Lua use require"iupluacontrols" or manually call the initialization function in C, **iupcontrolslua_open**, after calling **iuplua_open**.

When manually calling the function your application must be linked to the control library (**iupcontrols**), the CD_IUP driver (**iupcd**), and with the CD library (**cd**). To use its bindings to Lua, the program must also be linked to the **iupluacontrols** library.

IupCells

Creates a grid widget (set of cells) that enables several application-specific drawing, such as: chess tables, tiles editors, degrade scales, drawable spreadsheets and so forth.

This element is mostly based on application callbacks functions that determine the number of cells (rows and columns), their appearance and interaction. This mechanism offers full flexibility to applications, but requires programmers attention to avoid infinite loops inside this functions. Using callbacks, cells can be also grouped to form major or hierarchical elements, such as headers, footers etc. This callback approach was intentionally chosen to allow all cells to be dynamically and directly changed based on application's data structures. Since the size of each cell is given by the application the size of the control also must be given using SIZE or RASTERSIZE attributes.

This is an additional control that depends on the CD library. It is included in the IupControls library.

It inherits from IupCanvas.

Originally implemented by André Clinio.

Creation

```
Ihandle* IupCells(void); [in C]
iup.cells{} -> (ih: ihandle) [in Lua]
cells() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BOXED: Determines if the bounding cells' regions should be drawn with black lines. It can be "YES" or "NO". Default: "YES". If the span attributes are used, set this attribute to "NO" to avoid grid drawing over spanned cells.

BUFFERIZE: Disables the automatic redrawing of the control, so many attributes can be changed without many redraws. When set to "NO" the control is redrawn. When REPAINT attribute is set, BUFFERIZE is automatically set to "NO". Default: "NO".

CANVAS (read-only) (non inheritable): Returns the internal IUP CD canvas. This attribute should be used only in specific cases and by experienced CD programmers.

CLIPPED: Determines if, before cells drawing, each bounding region should be clipped. This attribute should be changed in few specific cases. It can be "YES" or "NO". Default: "YES".

FIRST_COL (read-only) (non inheritable): Returns the number of the first visible column.

FIRST_LINE (read-only) (non inheritable): Returns the number of the first visible line.

FULL_VISIBLE (write-only) (non inheritable)**:** Tries to show completely a specific cell (considering any vertical or horizontal header or scrollbar position) .This attribute is set by a formatted string "%d:%d" (C syntax), where each "%d" represent the line and column integer indexes respectively.

IMAGE_CANVAS (read-only) (non inheritable): Returns the internal image CD canvas. This attribute should be used only in specific cases and by experienced CD programmers.

LIMITS*L:C* (read-only) (non inheritable): Returns the limits of a given cell. Input format is "lin:col" or "%d:%d" in C. Output format is "xmin:xmax:ymin:ymax" or "%d:%d:%d:%d" in C.

NON_SCROLLABLE_LINES: Determines the number of non-scrollable lines (vertical headers) that should always be visible despite the vertical scrollbar position. It can be any non-negative integer value. Default: "0"

NON_SCROLLABLE_COLS: Determines the number of non-scrollable columns (horizontal headers) that should always be visible despite the horizontal scrollbar position. It can be any non-negative integer value. Default: "0"

ORIGIN: Sets the first visible line and column positions. This attribute is set by a formatted string "%d:%d" (C syntax), where each "%d" represent the line and column integer indexes respectively.

REPAINT(write-only) (non inheritable)**:** When set with any value, provokes the control to be redrawn.

SIZE (non inheritable): there is no initial size. You must define SIZE or RASTERSIZE.

SCROLLBAR (creation only): Default: "YES".

ACTIVE, BGCOLOR, FONT, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

DRAW_CB: called when a specific cell needs to be redrawn.

```
int function(Ihandle* ih, int line, int column, int xmin, int xmax, int ymin, int ymax, cdCanvas* canvas);  [in C]
ih:draw_cb(line, column, xmin, xmax, ymin, ymax: number, canvas: cdCanvas) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
line, **column**: the grid position inside the control that is being redrawn, in grid coordinates.
xmin, xmax, ymin, ymax: the raster bounding box of the redrawn cells, where the application can use CD functions to draw anything. If the attribute IUP_CLIPPED is set (the default), all CD graphical primitives is clipped to the bounding region.
canvas: internal canvas CD used to draw the cells.

HEIGHT_CB: called when the controls needs to know a (eventually new) line height.

```
int function(Ihandle* ih, int line);  [in C]
ih:height_cb(line: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
line: the line index

Returns: an integer that specifies the desired height (in pixels). Default is 30 pixels.

HSPAN_CB: called when the control needs to know if a cell should be horizontally spanned.

```
int function(Ihandle* ih, int line, int column);  [in C]
ih:hspan_cb(line, column: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
line, column: the line and column indexes (in grid coordinates)

Returns: an integer that specifies the desired span. Default is 1 (no span).

MOUSECLICK_CB: called when a color is selected. The primary color is selected with the left mouse button, and if existent the secondary is selected with the right mouse button.

```
int function(Ihandle* ih, int button, int pressed, int line, int column, int x, int y, char* status);  [in C]
ih:mouseclick_cb(button, pressed, line, column, x, y: number, string: status) -> (ret: number) [in Lua]
```

Same as the BUTTON_CB IupCanvas callback with two additional parameters:

line, **column**: the grid position in the control where the event has occurred, in grid coordinates.

MOUSEMOTION_CB: called when the mouse moves over the control.

```
int function(Ihandle *ih, int line, int column, int x, int y, char *r); [in C]
ih:mousemotion_cb(x, y: number, r: string) -> (ret: number) [in Lua]
```

Same as the MOTION_CB IupCanvas callback with two additional parameters:

line, **column**: the grid position in the control where the event has occurred, in grid coordinates.

NCOLS_CB: called when then controls needs to know its number of columns.

```
int function(Ihandle* ih);  [in C]
ih:ncols_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Returns: an integer that specifies the number of columns. Default is 10 columns.

NLINES_CB: called when then controls needs to know its number of lines.

```
int function(Ihandle* ih);  [in C]
ih:nlines_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Returns: an integer that specifies the number of lines. Default is 10 lines.

SCROLLING_CB: called when the scrollbars are activated.

```
int function(Ihandle* ih, int line, int column);  [in C]
ih:scrolling_cb(line, column: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
line, column: the first visible line and column indexes (in grid coordinates)

Returns: If IUP_IGNORE the cell is not redrawn. By default the cell is always redrawn.

VSPAN_CB: called when the control needs to know if a cell should be vertically spanned.

```
int function(Ihandle* ih, int line, int column);  [in C]
ih:vspan_cb(line, column: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
line, column: the line and column indexes (in grid coordinates)

Returns: an integer that specifies the desired span. Default is 1 (no span).

WIDTH_CB: called when the controls needs to know the column width

```
int function(Ihandle* ih, int column);  [in C]
ih:width_cb(column: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
column: the column index

Returns: an integer that specifies the desired width (in pixels). Default is 60 pixels.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Utility Functions

These functions can be used to help set and get attributes from the control:

```
void  IupSetAttributeId2(Ihandle* ih, const char* name, int lin, int col, const char* value);
char* IupGetAttributeId2(Ihandle* ih, const char* name, int lin, int col);
int   IupGetIntId2(Ihandle* ih, const char* name, int lin, int col);
float IupGetFloatId2(Ihandle* ih, const char* name, int lin, int col);
void  IupSetfAttributeId2(Ihandle* ih, const char* name, int lin, int col, const char* format, ...);
void  IupSetIntId2(Ihandle* ih, const char* name, int lin, int col, int value);
void  IupSetFloatId2(Ihandle* ih, const char* name, int lin, int col, float value);
```

```
IupSetAttribute(ih, "30:10", value)        => IupSetAttributeId2(ih, "", 30, 10, value)
IupSetAttribute(ih, "BGCOLOR30:10", value) => IupSetAttributeId2(ih, "BGCOLOR", 30, 10, value)
IupSetAttribute(ih, "ALIGNMENT10", value)  => IupSetAttributeId(ih, "ALIGNMENT", 10, value)
```

When one of the indices is the asterisk, use IUP_INVALID_ID as the parameter. For ex:

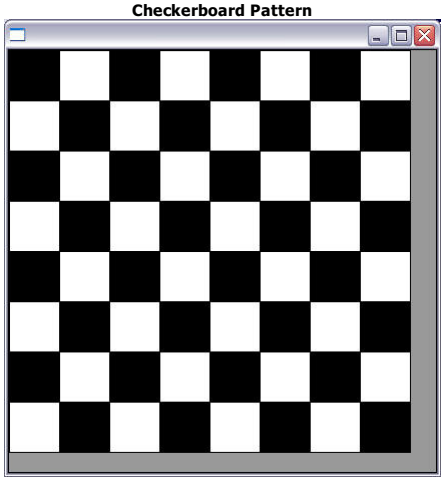
```
IupSetAttribute(ih, "BGCOLOR30:*", value) => IupSetAttributeId2(ih, "BGCOLOR", 30, IUP_INVALID_ID, value)
```

These functions are faster than the traditional functions because they do not need to parse the attribute name string and the application does not need to concatenate the attribute name with the id.

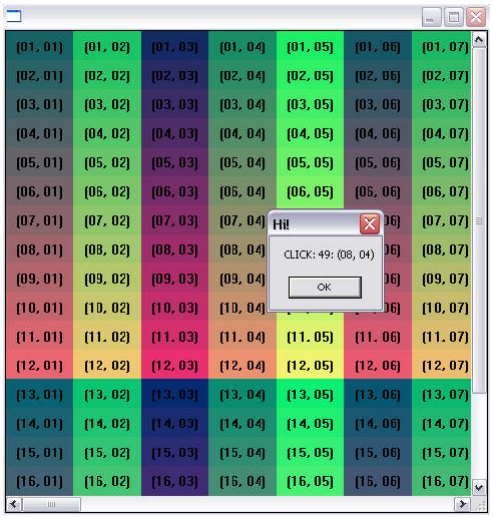
Examples

Browse for Example Files

Checkerboard Pattern



Numbering Cells



See Also

IupCanvas

IupColorbar

Creates a color palette to enable a color selection from several samples. It can select one or two colors. The primary color is selected with the left mouse button, and the secondary color is selected with the right mouse button. You can double click a cell to change its color and you can double click the preview area to switch between primary and secondary colors.

This is an additional control that depends on the CD library. It is included in the IupControls library.

It inherits from IupCanvas.

Originally implemented by André Clinio.

Creation

```
Ihandle* IupColorbar(void); [in C]
iup.colorbar{} -> (ih: ihandle) [in Lua]
colorbar() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BUFFERIZE (non inheritable)**:** Disables the automatic redrawing of the control, so many attributes can be changed without many redraws. Default: "NO". When set to "NO" the control is redrawn.

CELL*n*: Contains the color of the "n" cell. "n" can be from 0 to NUM_CELLS-1.

NUM_CELLS (non inheritable)**:** Contains the number of color cells. Default: "16". The maximum number of colors is 256. The default colors use the same set of **IupImage**.

COUNT (read-only) (non inheritable)**:** same as **NUM_CELLS** but it is read-only. (since 3.3)

NUM_PARTS (non inheritable)**:** Contains the number of lines or columns. Default: "1".

ORIENTATION: Controls the orientation. It can be "VERTICAL" or "HORIZONTAL". Default: "VERTICAL".

PREVIEW_SIZE (non inheritable)**:** Fixes the size of the preview area in pixels. The default size is dynamically calculated from the size of the control. The size is reset to the default when SHOW_PREVIEW=NO.

SHOW_PREVIEW: Controls the display of the preview area. Default: "YES".

SHOW_SECONDARY: Controls the existence of a secondary color selection. Default: "NO".

SIZE: there is no initial size. You must define SIZE or RASTERSIZE.

PRIMARY_CELL (non inheritable)**:** Contains the index of the primary color. Default "0" (black).

SECONDARY_CELL (non inheritable)**:** Contains the index of the secondary color. Default "15" (white).

SQUARED: Controls the aspect ratio of the color cells. Non square cells expand equally to occupy all of the control area. Default: "YES".

SHADOWED: Controls the 3D effect of the color cells. Default: "YES".

TRANSPARENCY: Contains a color that will be not rendered in the color palette. The color cell will have a white and gray chess pattern. It can be used to create a palette with less colors than the number of cells.

ACTIVE, BGCOLOR, FONT, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, EXPAND, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

CELL_CB: called when the user double clicks a color cell to change its value.

```
char* function(Ihandle* ih, int cell); [in C]
ih:cell_cb(cell: number) -> (ret: string) [in Lua]
```

ih: identifier of the element that activated the event.
cell: index of the selected cell. If the user double click a preview cell, the respective index is returned.

Returns: a new color or NULL (nil in Lua) to ignore the change. By default nothing is changed.

EXTENDED_CB: called when the user right click a cell with the Shift key pressed. It is independent of the SHOW_SECONDARY attribute.

```
int function(Ihandle* ih, int cell);  [in C]
ih:extended_cb(cell: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
cell: index of the selected cell.

Returns: If IUP_IGNORE the cell is not redrawn. By default the cell is always redrawn.

SELECT_CB: called when a color is selected. The primary color is selected with the left mouse button, and if existent the secondary is selected with the right mouse button.

```
int function(Ihandle* ih, int cell, int type);  [in C]
ih:select_cb(cell, type: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
cell: index of the selected cell.
type: indicates if the user selected a primary or secondary color. In can be: `IUP_PRIMARY(-1)` or `IUP_SECONDARY(-2)`.

Returns: If IUP_IGNORE the selection is not accepted. By default the selection is always accepted.

SWITCH_CB: called when the user double clicks the preview area outside the preview cells to switch the primary and secondary selections. It is only called if SHOW_SECONDARY=YES**.**

```
int function(Ihandle* ih, int prim_cell, int sec_cell);  [in C]
ih:switch_cb(prim_cell, sec_cell: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
prim_cell: index of the actual primary cell.
sec_cell: index of the actual secondary cell.

Returns: If IUP_IGNORE the switch is not accepted. By default the switch is always accepted.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

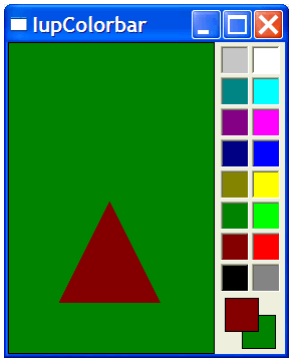
Notes

When the control has the focus the keyboard can be used to change the colors and activate the callbacks. Use the arrow keys to move from cell to cell, **Home** goes to the first cell, **End** goes to the last cell. **Space** will activate the **SELECT_CB** callback for the primary color, **Ctrl**+**Space** will activate the **SELECT_CB** callback for the secondary color. **Shift**+**Space** will activate the **EXTENTED_CB** callback. **Shift**+**Enter** will activate the **CELL_CB** callback.

Examples

Browse for Example Files

Creates a Colorbar for selection of two colors.



See Also

IupCanvas, IupImage

Attributes

EXPAND: The default is "NO".

RASTERSIZE (non inheritable): the initial size is "181x181". Set to NULL to allow the automatic layout use smaller values.

RGB (non inheritable): the color selected in the control, in the "r g b"Â format; r, g and b are integers ranging from 0 to 255. Default: "255 0 0".

HSI (non inheritable): the color selected in the control, in the "h s i"Â format; h, s and i are floating point numbers ranging from 0-360, 0-1 and 0-1 respectively.

ACTIVE, BGCOLOR, FONT, X, Y, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, ZORDER, VISIBLE: also accepted.

Callbacks

CHANGE_CB: Called when the user releases the left mouse button over the control, defining the selected color.

```
int change(Ihandle *ih, unsigned char r, unsigned char g, unsigned char b); [in C]
ih:change_cb(r: number, g: number, b: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
r, g, b: color value.

DRAG_CB: Called several times while the color is being changed by dragging the mouse over the control.

```
int drag(Ihandle *ih, unsigned char r, unsigned char g, unsigned char b); [in C]
ih:drag_cb(r: number, g: number, b: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
r, g, b: color value.

VALUECHANGED_CB: Called after the value was interactively changed by the user. It is called whenever a **CHANGE_CB** or a **DRAG_CB** would also be called, it is just called after them. (since 3.0)

```
int function(Ihandle *ih); [in C]
ih:valuechanged_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

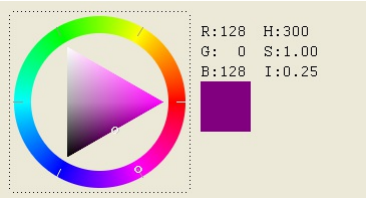
When the control has the focus the keyboard can be used to change the color value. Use the arrow keys to move the cursor inside the SI triangle, and use Home(0), PageUp, PageDn and End(180) keys to move the cursor inside the Hue circle.

The Hue in the HSI coordinate system defines a plane that it is a triangle in the RGB cube. But the maximum saturation in this triangle is different for each Hue because of the geometry of the cube. In ColorBrowser this point is fixed at the center of the **I** axis. So the **I** axis is not completely linear, it is linear in two parts, one from 0 to 0.5, and another from 0.5 to 1.0. Although the selected values are linear specified you can notice that when Hue is changed the gray scale also changes, visually compacting values above or below the I=0.5 line according to the selected Hue.

This is the same HSI specified in the IM toolkit, except for the non linearity of **I**. This non linearity were introduced so a simple triangle could be used to represent the SI plane.

Examples

Browse for Example Files



See Also

IupGetColor, IupColorDlg.

ih: identifier of the element that activated the event.
angle: the dial value converted according to UNIT.

VALUECHANGED_CB: Called after the value was interactively changed by the user. It is called whenever a **BUTTON_PRESS_CB**, a **BUTTON_RELEASE_CB** or a **MOUSEMOVE_CB** would also be called, but if defined those callbacks will not be called. (since 3.0)

```
int function(Ihandle *ih); [in C]
ih:valuechanged_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

When the keyboard arrows are pressed and released the mouse press and the mouse release callbacks are called in this order. If you hold the key down the mouse move callback is also called for every repetition.

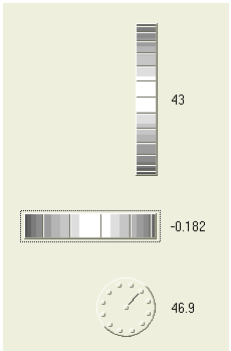
When the wheel is rotated only the mouse move callback is called, and it increments the last angle the dial was rotated.

In all cases the value is incremented or decremented by PI/10 (18 degrees).

If you press Shift while using the arrow keys the increment is reduced to PI/100 (1.8 degrees). Press the Home key in the circular dial to reset to 0. (since 3.0)

Examples

Browse for Example Files



See Also

IupCanvas

IupGauge

Creates a Gauge control. Shows a percent value that can be updated to simulate a progression. It inherits from IupCanvas.

This is an additional control that depends on the CD library. It is included in the IupControls library.

It is recommended that new applications use the IupProgressBar control of the main library.

Creation

```
Ihandle* IupGauge(void); [in C]
iup.gauge{} -> (ih: ihandle) [in Lua]
gauge() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

DASHED: Changes the style of the gauge for a dashed pattern. Default is "NO".

FGCOLOR: Controls the gauge and text color. The default is "64 96 192".

MAX (non inheritable): Contains the maximum value. Default is "1".

MIN (non inheritable): Contains the minimum value. Default is "0".

PADDING: internal margin. Works just like the MARGIN attribute of the **IupHbox** and **IupVbox** containers, but uses a different name to avoid inheritance problems. Default value: "0x0". (since 3.0)

SHOWTEXT: Indicates if the text inside the Gauge is to be shown or not. If the gauge is dashed the text is never shown. Possible values: "YES" or "NO". Default: "YES".

SIZE (non inheritable): The initial size is "120x14". Set to NULL to allow the automatic layout use smaller values.

TEXT (non inheritable): Contains a text to be shown inside the Gauge when SHOW_TEXT=YES. If it is NULL, the percentage calculated from VALUE will be used. If the gauge is dashed the text is never shown.

VALUE (non inheritable): Contains a number between "MIN" and "MAX", controlling the current position.

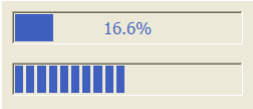
ACTIVE, BGCOLOR, EXPAND, FONT, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

MAP_CB, UNMAP_CB, DESTROY_CB: common callbacks are supported.

Examples

Browse for Example Files



The Two Types of Gauge

See Also

IupCanvas

action_cb: Name of the action generated when the user types something.

Returns the identifier of the created matrix, or NULL if an error occurs.

Attributes

See General Attributes

See Cell Attributes

See Column Attributes

See Size Attributes

See Column Size Attributes

See Line Size Attributes

See Number of Cells Attributes

See Mark Attributes Attributes

See Action Attributes Attributes

See Editing Attributes Attributes

See Text Editing Attributes Attributes

See Canvas Attributes Attributes

Callbacks

Interaction

ACTION_CB - Action generated when a keyboard event occurs.
CLICK_CB - Action generated when any mouse button is pressed over a cell.
COLRESIZE_CB: Action generated when a column is interactively resized.
RELEASE_CB - Action generated when any mouse button is released over a cell.
RESIZEMATRIX_CB: Action generated after the element size has been updated but before the cells have been actually refreshed.
TOGGLEVALUE_CB: Action generated when a toggle button is pressed.
VALUECHANGED_CB: Called after the value was interactively changed by the user or after a group of values where programmatically changed in a single operation.
MOUSEMOVE_CB - Action generated to notify the application that the mouse has moved over the matrix.

ENTERITEM_CB - Action generated when a matrix cell is selected, becoming the current cell.
LEAVEITEM_CB - Action generated when a cell is no longer the current cell.
SCROLLTOP_CB - Action generated when the matrix is scrolled with the scrollbars or with the keyboard.

Drawing

BGCOLOR_CB - Action generated to retrieve the background color of a cell when it needs to be redrawn.
FGCOLOR_CB - Action generated to retrieve the foreground color of a cell when it needs to be redrawn.
FONT_CB - Action generated to retrieve the font of a cell when it needs to be redrawn.
TYPE_CB: Action generated to retrieve the type of a cell value.
DRAW_CB - Action generated before the cell is drawn. Allow a custom cell draw.
DROPCHECK_CB - Action generated to determine if a dropdown feedback should be shown.
TRANSLATEVALUE_CB: Action generated to translate the value of a cell during display and size computation.

Editing

DROP_CB - Action generated to determine if a text field or a dropdown will be shown.
MENUDROP_CB - Action generated to determine if a popup menu will be shown.
DROPSELECT_CB - Action generated when an element in the dropdown list is selected.
EDITION_CB - Action generated when the current cell enters or leaves the edition mode.

Callback Mode

VALUE_CB - Action generated to verify the value of a cell.
VALUE_EDIT_CB - Action generated to notify the application that the value of a cell was edited.
MARK_CB - Action generated to verify the selection state of a cell.
MARKEDIT_CB - Action generated to notify the application that the selection state of a cell was changed.

Utility Functions

These functions can be used to help set and get attributes from the matrix:

```
void  IupSetAttributeId2(Ihandle* ih, const char* name, int lin, int col, const char* value);
char* IupGetAttributeId2(Ihandle* ih, const char* name, int lin, int col);
int   IupGetIntId2(Ihandle* ih, const char* name, int lin, int col);
float IupGetFloatId2(Ihandle* ih, const char* name, int lin, int col);
void  IupSetfAttributeId2(Ihandle* ih, const char* name, int lin, int col, const char* format, ...);
void  IupSetIntId2(Ihandle* ih, const char* name, int lin, int col, int value);
void  IupSetFloatId2(Ihandle* ih, const char* name, int lin, int col, float value);
```

They work just like the respective traditional set and get functions. But the attribute string is complemented with the L and C values. When only one value is needed then use the Iup*AttributeId functions. For ex:

```
IupSetAttribute(ih, "30:10", value)        => IupSetAttributeId2(ih, "", 30, 10, value)
IupSetAttribute(ih, "BGCOLOR30:10", value) => IupSetAttributeId2(ih, "BGCOLOR", 30, 10, value)
IupSetAttribute(ih, "ALIGNMENT10", value)  => IupSetAttributeId(ih, "ALIGNMENT", 10, value)
```

When one of the indices is the asterisk, use IUP_INVALID_ID as the parameter. For ex:

```
IupSetAttribute(ih, "BGCOLOR30:*", value) => IupSetAttributeId2(ih, "BGCOLOR", 30, IUP_INVALID_ID, value)
```

These functions are faster than the traditional functions because they do not need to parse the attribute name string and the application does not need to concatenate the attribute name with the id.

They are used by the additional methods in Lua:

```
ih:setcell(lin, col: number, value: string)
ih:getcell(lin, col: number) -> (cell: string)
```

But you can also use the traditional functions when typing:

```
ih["bgcolor"..l.."":"..c] = v
  or
ih["bgcolor30:10"] = v
```

```
void IupMatrixSetFormula(Ihandle* ih, int col, const char* formula, const char* init); [in C]
iup.MatrixSetFormula(ih: ihandle, col: number, formula: string, [init: string]) [in Lua]
or ih:SetFormula(col: number, formula: string, [init: string]) [in Lua]
```

Fill the contents of the given column using the formula (since 3.13). The formula is executed for each line within the column. Internally uses Lua to parse the formula. **init** is an optional Lua initialization code that is called only once (can be NULL). The callback "int **FORMULAINIT_CB**(Ihandle* ih, lua_State *L);" can also be used to initialize the Lua state. All Lua standard libraries are pre-loaded.

This function is available in the "iupluacontrols" library but it does not requires an active Lua context, because it uses a temporary Lua context. If called from Lua it will also be independent from the application's Lua context. To use it in C/C++ you must link also with Lua and iuplua even when not using theses libraries directly.

The formula will be encapsulated within an internal Lua function so it will not affect the call of subsequent cells. This internal function receives two parameters "lin" and "col" correspondent to the current cell being processed during script execution. The formula can contain only one valid Lua statement that will be retuned by the internal Lua function. The formula can evaluate to nil, number, boolean or a string.

The most commonly used tokens are:

```
+   (addition)
-   (subtraction and negation)
*   (multiplication)
/   (division)
%   (modulo)
^   (exponentiation)
==  (equal)
~=  (different)
<   (less than)
>   (greater than)
<=  (less than or equal)
>=  (greater than or equal)
and (logical and)
or  (logical or)
not (logical not)
```

The Lua Math Functions are loaded also at the global level, so there is no need for the "math." prefix. The most commonly used functions are:

```
abs(x)    acos(x)    asin(x)    atan(x)    atan2(y,x)    ceil(x)    cos(x)
deg(x)    exp(x)     floor(x)   log(x)     min(x,...)    max(x,...) pow(x,y)
sin(x)    sqrt(x)    tan(x)
```

There are also some exclusive functions to access cell values and perform special operations:

```
sum(x,...)  - computes the sum of the input paramters.
average(x,...)  - computes the average of the input paramters.
range(lin1, col1, lin2, col2[, only_number]) - returns a range of cell values.
                    Can be used in functions like min, max, sum and average.
                    If only_number boolean is used then only numbers are included,
                    and others are skipped.
cell(lin, col) - returns the cell value at given line and column.
ifelse(test, value_true, value_false) - if test boolean is true then return value_true,
                                        if not return value_falseor else.
      The problem with ifelse is that both values are evaluated before calling the function.
      In Lua the solution is to use logical operators:
      test and value_true or value_false
      (but value_true can not be false)
      See http://lua-users.org/wiki/TernaryOperator
```

If the attribute CELLNAMES is set to "Excel" or "Matrix" (default "No") then it will enable cell names to be used as alternative for "cell(lin, col)". There are two notations available: the Matrix "L123C123" notation where L and C are fixed and 123s are the line and columns numbers; and the Excel "ABC123" notation where 123 denotes the line number and ABC denotes the column number just like in Microsoft Excel. (since 3.14) Obs: there is not support for cell range like "A1:B2" in Excel.

Some formula examples:

```
"cos(pi*lin/4)"
"cell(lin, 1) + cell(lin, 2)"
"cell(lin, 4) < 3"  -- cell value will be 0 or 1
"sum(range(lin, 1, lin, 7))"
"cell('x', 1)"  -- error
```

```
void IupMatrixSetDynamic(Ihandle* ih, const char* init); [in C]
iup.MatrixSetDynamic(ih: ihandle, [init: string]) [in Lua]
or ih:SetDynamic([init: string]) [in Lua]
```

Enable dynamic cell values using formulas (since 3.13). It uses the TRANSLATEVALUE_CB callback to process strings just before the value is displayed, if the string starts with a equal sign ("=") then it is interpreted as a formula using the same features and rules as the **IupMatrixSetFormula** function above. If the value is being edited the callback will return the original value so the formula can be edited.

Internally also uses Lua to parse the formula. **init** is an optional Lua initialization code that is called only once (can be NULL), at the function call. The callback "int **FORMULAINIT_CB**(Ihandle* ih, lua_State *L);" can also be used to initialize the Lua state, at the function call. All Lua standard libraries are pre-loaded.

This Lua state is initialize at the function call and saved for processing during the TRANSLATEVALUE_CB callback. If IupMatrixSetDynamic is called again then the previous state will be destroyed and a new one will be created. This state is automatically destroyed when the control is destroyed.

This function is available in the "iupluacontrols" library but it does not requires an active Lua context, because it uses a temporary Lua context. If called from Lua it will also be independent from the application's Lua context. To use it in C/C++ you must link also with Lua and iuplua even when not using theses libraries directly.

If the cell has a formula, i.e. starts with the equal sign, and the attribute EDITHIDEONFOCUS is NO, then during editing the user can click on another cell to insert a reference to its value in the format "cell(lin,col)". If CELLNAMES is enabled then the respective cell name will be used instead of the "cell" function call. Selecting a range of cells it will insert a "range(lin1,col1,lin2,col2)" call instead (there is no special notation for a range). (since 3.14)

Notes

Storage

Before mapped to the native system, all attributes are stored in the hash table, independently from the size of the matrix or its operation mode. The action attributes like ADDLIN and DELCOL will NOT work.

When the matrix is mapped, and it is NOT in callback mode, then the cell values and mark state are moved from the hash table to an internal storage at the matrix. Other cell attributes remains on the hash table. Cell values with indices greater than (NUMLIN,NUMCOL) are ignored. When in callback mode cell values stored in the hash table are ignored.

Callback Mode

Very large matrices can use the callback mode to store the cell values at the application, and not at the internal matrix storage. The idea is the following:

1 - Register the VALUE_CB callback
2 - No longer set the value of the cells. Store the cell value at the application. They will be retrieved one by one by the callback.
3 - If the matrix can be edited, set the VALUE_EDIT_CB callback.
4 - When the matrix display must be updated, use the REDRAW attribute to force a matrix redraw.

A negative aspect is that, when VALUE_CB is defined, after it is mapped the matrix never verifies attributes of type L:C again.

If VALUE_CB is defined and VALUE_EDIT_CB is not defined when the matrix is mapped then READONLY will be set to YES.

Number of Cells

If you do not plan to use ADDLIN nor ADDCOL, and plan to set sparse cell values, then you must set NUMLIN and NUMCOL before mapping.

Titles

A matrix might have titles for lines and columns. Titles are always non scrollable, non editable and presented with a different default background color. A matrix will have a line of titles if an attribute of the "*L*:0" type is defined, where L is a line number, or if the HEIGHT0 attribute is defined. It will have a column of titles if an attribute of the "0:*C*" type is defined, where C is a column number, or if the WIDTH0 attribute is defined.

When allowed the width of a column can be changed by holding and dragging its title right border, see RESIZEMATRIX.

Any cell can have more than one text line, just use the \n control character. Multiple text lines will be considered when calculating the title cell size based on its contents. The contents of ordinary cells (not a title) do not affect the cell size.

Natural Size

The Natural size is calculated using only the title cells size plus the size of NUMCOL_VISIBLE and NUMLIN_VISIBLE cells, but it is also affected if SCROLBAR is enabled. The natural height is the sum of the line heights from line 0 to NUMLIN_VISIBLE (inclusive). The natural width is the sum of the column width from column 0 to NUMCOL_VISIBLE (inclusive). Notice that since NUMCOL_VISIBLE and NUMLIN_VISIBLE do not include the titles then NUMCOL_VISIBLE+1 columns and NUMLIN_VISIBLE+1 lines are included in the sum.

The height of a line L depends on several attributes, first it checks the HEIGHT*L* attribute, then checks RASTERHEIGHT*L*, then when USETITLESIZE=YES or not in callback mode the height of the title text for the line or if L=0 it searches for the highest column title, if still could not define a height then if L!=0 it will use HEIGHTDEF, if L=0 then height will be 0.

A similar approach is valid for the column width. The width of a column C first checks the WIDTH*C* attribute, then checks RASTERWIDTH*C*, then when USETITLESIZE=YES or not in callback mode the width of the title text for the column or if C=0 it searches for the widest line title, if still could not define a width then if C!=0 it will use WIDTHDEF, if C=0 then height will be 0.

Virtual Size

When the scrollbars are enabled if the matrix area is greater than the visible area then scrollbars will be displayed so the cells can be scrolled to be visible area. When dragging the scrollbar the position of cells is free, when clicking on its buttons it will move in cell steps, aligning to the left border of the cell.

By default EXPAND=Yes, so matrix will be automatically resized when the dialog is resized. So more columns and lines will be displayed. But the matrix Natural size will be used as minimum size. To remove the minimum size limitation set NUMCOL_VISIBLE and NUMLIN_VISIBLE to 0 after showing it for the first time.

Edition Mode

When READONLY=NO and there is no EDITION_CB callback or the callback return is IUP_DEFAULT, the matrix cell values can be edited.

Editing starts automatically when the user press a character key when the focus is at a cell, then the old cell value is replaced by the new one being typed. If **F2, Enter** or **Space** is pressed, the current cell enters the edition mode with the current text of the cell. And double-clicking a cell enters the edition mode (in Motif the user must click again to the edit control get the focus).

The new value will be accepted if the user press **Enter** during edition mode. Pressing **Esc** will cancel the editing and the the old value remains. The cell will also leave the edition mode if the user clicked in another cell or in another control, then the new value will be automatically accepted. But the value confirmation still depends on the EDITION_CB callback return code.

Keyboard Navigation

Keyboard navigation through the matrix cells outside the edition mode is done by using the following keys:

- **Arrows**: Moves the focus to the next cell, according to the arrows direction.
- **Page Up** and **Page Down**: Moves a visible page up or down.
- **Home**: Moves the focus to the fist column in the line.
- **Home Home**: Moves the focus to the upper left corner of the visible page.
- **Home Home Home**: Moves the focus to the upper left corner of the first page of the matrix.
- **End**: Moves the focus to the last column in the line.
- **End End**: Moves the focus to the lower right corner of the visible page.
- **End End End**: Moves the focus to the lower right corner of the last page in the matrix.

When using the keyboard to change the focus cell if the limit of the visible area is reached then the cells are automatically scrolled. Also if a cell partially visible is edited then first it is scrolled to the visible area. Also while pressing together the **Shift** key and marks are enabled with MARKMULTIPLE=Yes then a continuous area will be selected (since 3.9).

Inside the **edition mode**, the following keys are used for a text field:

- **Left, Right, Up and Down arrows**: if the caret is at the extremes of the text being edited then leave the edition mode and moves the focus accordingly. The value is confirmed.
- **Ctrl + arrows**: leave the edition mode and moves the focus accordingly independent of caret position. The value is confirmed.
- **Enter**: leave the edition mode. The value is confirmed. Moves the focus to the cell below.
- **Esc**: leave the edition mode. The new value is ignored and the old value remains.

When pressing **Enter** to confirm the value the focus goes to the cell below the current cell, if at the last line then the focus goes to the cell on the left. This can be controlled using the EDITNEXT attribute.

Marks (Selected Cells)

When a mark mode is set the cells can be marked using mouse.

A marked cell will have its background attenuated to indicate that it is marked. A title cell appears marked only when MARKMODE=LIN, COL or LINCOL.

Cells can be selected individually or can be restricted to lines or columns. Also multiple cells can be marked simultaneously in continuous or in segmented areas. Lines and columns are marked only when the user clicks in their respective titles, if MARKMODE=CELL then all the cells of the line or column will be marked. Continuous areas are marked holding and dragging the mouse or holding the **Shift** key when clicking at the end of the area. Segmented areas are marked or unmarked holding the **Ctrl** key, the mark state is inverted. Clicking on the cell 0:0 will select all the items depending on MARKMODE, except for LINCOL.

When there are cells marked, pressing the **Del** key remove the selected cells contents.

IupMatrixEx

For more features, like Import/Export, Clipboard, Undo/Redo, Search, Sort, Column Visibility, Numeric Columns, Numeric, Context Menu and others, see the IupMatrixEx extension library.

Examples

Browse for Example Files

Inflation	January 2000	February 2000
Medicine	5.6	4.5
Pharma	3.33	
Food	2.2	8.1
Energy	7.2	3.4

1:1	1:2	1:3
2:1	2:2	2:3
3:1	3:2	3:3

See Also

IupCanvas, IupMatrixEx

IupMatrix Attributes (all non inheritable, with exceptions)

General Attributes

CURSOR: Default cursor used by the matrix. The default cursor is a symbol that looks like a cross. If you need to refer to this default cursor, use the name "IupMatrixCrossCursor".

DROPIMAGE: drop image name. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. By default an internal image will be used. (since 3.16)

FOCUSCELL: Defines the current cell. Two numbers in the "***L***:***C***" format, (***L***>0 and ***C***>0, a title cell can NOT be the current cell). Default: "1:1".

HIDEFOCUS: do not show the focus mark when drawing the matrix. Default is NO.

HIDDENTEXTMARKS: when text is greater than cell space, it is normally cropped, but when set to YES a "..." mark will be added at the crop point to indicate that there is more text not visible.

Default: NO. (since 3.1)

HLCOLOR (non inheritable): the overlay color for the selected cells. Default: TXTHLCOLOR global attribute. If set to "" will only use the attenuation process. The color is composited using HLCOLORALPHA attribute as alpha value (default is 128). (since 3.16)

ORIGIN: Scroll the visible area to the given cell. Returns the cell at the upper left corner. To scroll to a line or a column, use a value such as "***L***:*" or "*:***C***" (where ***L***>0 and ***C***>0). L and C can not be a non scrollable cell either.

ORIGINOFFSET: complements the ORIGIN attribute by specifying the drag offset of the top left cell. Returns the current value. Has the format "X:Y" or "%d:%d" in C. When changing this attribute must change also ORIGIN right after. (since 3.5)

READONLY: disables the editing of all cells. EDITION_CB and VALUE_EDIT_CB will not be called anymore. The L:C attribute will still be able to change the cell value. (since 3.0)

SHOWFILLVALUE: enable the display of the numeric percentage in the cell when TYPE* is FILL. Default: NO. (since 3.9)

TOGGLECENTERED: center the toggle and use the cell value in place of TOGGLEVALUE*L:C*. No text will be drawn. (since 3.16)

TOGGLEIMAGEON/TOGGLEIMAGEOFF: sort sign image name. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. By default an internal image will be used. (since 3.16)

ACTIVE, EXPAND, FONT, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Cell Attributes (no redraw)

(These attributes are only updated in the display when you set the REDRAW attribute.)

L:C: Text of the cell located in line L and column C, where L and C are integer numbers.
***L*:0**: Title of line L.
0:*C*: Title of column C.
0:0: Title of the area between the line and column titles.

These are valid only in normal mode.

ALIGN*L:C*: Alignment of the cell value in line L and column C. Values are in the format: "linalign:colalign", where linalign can be "ATOP", "ACENTER" or "ABOTTOM", and colalign can be "ALEFT", "ACENTER" or "ARIGHT". Default will use ALIGNMENT* and LINEALIGMENT*. (Since 3.16)

TYPE*L:C*: Type of the cell value in line L and column C. (Since 3.9)
TYPE*:C*: Type of column C. (Since 3.9)
TYPE*L*:*: Type of line L. (Since 3.9)

Can be TEXT, COLOR, FILL, or IMAGE. When type is COLOR the cell value is interpreted as a color and a rectangle with the color is drawn inside the cell instead of the text (the FGCOLOR of the cell is ignored). When type is FILL the cell value is interpreted as percentage and a rectangle showing the percentage in the FGCOLOR is drawn like in **IupGauge** and **IupProgressBar**. When type is IMAGE the cell value is interpreted as an image name, and if an image exist with that name is drawn (the name can NOT be of a Windows resource or GTK stock image). Only TEXT and IMAGE are affected by alignment attributes. Default: TEXT. (Since 3.9)

BGCOLOR: Background color of the matrix. (inheritable)
BGCOLOR*:C*: Background color of column C.
BGCOLOR*L*:*: Background color of line L.
BGCOLOR*L:C*: Background color of the cell in line L and column C.

When more than one attribute are defined, the background color will be selected following this priority: BGCOLORL:C, BGCOLORL:*, BGCOLOR*:C, and last BGCOLOR. (L or C >= 0)
Default BGCOLOR is the global attribute TXTBGCOLOR for cells and the parent's BGCOLOR for titles.
Since the matrix control can be larger than the matrix itself, the empty area will always be filled with the parent's BGCOLOR.

FGCOLOR: Text color. (inheritable)
FGCOLOR*:C*: Text color of column C.
FGCOLOR*L*:*: Text color of line L.
FGCOLOR*L:C*: Text color of the cell in line L and column C.

When more than one attribute are define, the text color of a cell will be selected following this priority: FGCOLORL:C, FGCOLORL:*, FGCOLOR*:C, and last FGCOLOR. (L or C >= 0)
Default FGCOLOR is the global attribute TXTFGCOLOR for cells or the global attribute DLGFGCOLOR for titles.

FONT: Character font of the text. (inheritable)
FONT*L*:*: Text font of the cells in line L.
FONT*:C*: Text font of the cells in column C.
FONT*L:C*: Text font of the cell in line L and column C.

This attribute must be set before the control is showed. It affects the calculation of the size of all the matrix cells. The cell size is always calculated from the base FONT attribute.

FRAMECOLOR: Sets the color to be used in the frame lines. (inheritable)
FRAMEVERTCOLOR*L:C*: Color of the vertical right frame line of the cell. When not defined the FRAMECOLOR is used. For a title column cell (col=0) defines right and left frames. If value is "BGCOLOR" the frame line is not drawn.
FRAMEVERTCOLOR*:C*: same as FRAMEVERTCOLOR*L:C* but for all the cells of the column C. (since 3.5)
FRAMEHORIZCOLOR*L:C*: Color of the horizontal bottom frame line of the cell. When not defined the FRAMECOLOR is used. For a title line cell (lin=0) defines bottom and top frames. If value is "BGCOLOR" the frame line is not drawn.
FRAMEHORIZCOLOR*L:**:same as FRAMEHORIZCOLOR*L:C* but for all the cells of the line L. (since 3.5)

FRAMETITLEHIGHLIGHT: by default the title cells will have a bright line at left and top to configure a raise appearance. Can be Yes or No. Default: Yes. (since 3.9)

RESIZEMATRIXCOLOR: color used by the column resize feedback. Default: "102 102 102". (Since 3.9)

TOGGLEVALUE*L:C*: value of the toggle inside the cell. The toggle is shown only if the DROPCHECK_CB returns IUP_CONTINUE for the cell. When the toggle is interactively change the TOGGLEVALUE_CB callback is called. (Since 3.9)

VALUE: Allows setting or verifying the value of the current cell. Is the same as obtaining the current cell line and column from FOCUSCELL attribute, and then using them to access the "L:C" attribute. But when updated or retrieved during cell editing, the edit control will be updated or consulted instead of the matrix cell. When retrieved inside the EDITION_CB callback when mode is 0, then the return value is the new value that will be updated in the cell.

CELL*L:C* (read-only): Returns the displayed cell value. Returns NULL if the cell does not exists, or it is not visible, or the element is not mapped. (since 3.14)

CELLBGCOLOR*L:C* (read-only): Returns the actual cell background color, including mark and active state modifications. Returns NULL if the cell does not exists, or it is not visible, or the element is not mapped. (since 3.6)

CELLFGCOLOR*L:C* (read-only): Returns the actual cell foreground color, including mark state modifications. Returns NULL if the cell does not exists, or it is not visible, or the element is not mapped. (since 3.6)

CELLOFFSET*L:C* (read-only): Returns the cell computed offset in pixels from the top-left corner of the matrix, in the format "XxY" or "%dx%d" in C. Returns NULL if the cell does not exists, or it is not visible, or the element is not mapped. It will only return a valid result if the cell has already been displayed. They are similar to the parameters of the DRAW_CB callback but they do NOT include the decorations. (since 3.5)

CELLSIZE*L:C* (read-only): Returns the cell computed size in pixels, in the format "WxH" or "%dx%d" in C. Returns NULL if the cell does not exists, or the element is not mapped. It will only return a valid result if the cell has already been displayed. They are similar to the parameters of the DRAW_CB callback but they do NOT include the decorations. (since 3.5)

Column/Line Only Attributes (no redraw)

ALIGNMENT*C*: Horizontal alignment of the cells in column C (C >= 0) for lines that greater than 0. Can be: "ALEFT", "ACENTER" or "ARIGHT". Default: "ALEFT" for C=0 and "ACENTER" for C>0. Before checking the default value it will check the "**ALIGNMENT**" attribute value. If the text do not fit in the cell then the alignment is changed to ALEFT.

ALIGNMENTLIN0: Horizontal alignment of all the cells in line 0. Default is "ACENTER". (since 3.9)

LINEALIGNMENT*L*: Vertical alignment of the cells in line L (L >= 0) for all columns. Can be: "ATOP", "ACENTER" or "ABOTTOM". Default is "ACENTER". (since 3.16)

SORTSIGN*C* : Shows a sort sign (up or down arrow) in the column C (C >= 0) title. Possible values: "UP", "DOWN" and "NO". Default: NO.

SORTIMAGEDOWN/SORTIMAGEUP: sort sign image name. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. By default an internal image will be used. (since 3.16)

Size Attributes

LIMITEXPAND: limit expansion to the maximum size that shows all cells. This will set the MAXSIZE attribute to match the natural size of the matrix when all cells are visible. When the scrollbars have *AUTOHIDE=Yes, the maximum size will not include the scrollbars (since 3.9). (since 3.5)

RESIZEMATRIX: Defines if the width of a column can be interactively changed. When this is possible, the user can change the size of a column by dragging the column title right border. Possible values: "YES" or "NO". Default: "NO" (does not allow interactive width change).

USETITLESIZE: Use the title size to define the cell size if necessary. See WIDTHn and HEIGHTn. Default: NO. (since 3.0)

Column Size Attributes

For all columns if WIDTHn is not defined, then RASTERWIDTHn is used. If also not defined, then depending on the circumstances a logic is used to find the column width.

If it is the title column (n=0), then if USETITLESIZE=YES or not in callback mode, it will search for the maximum width among the titles of all lines. Finally if the conditions are not true or the maximum width of the column is 0, then the column of line titles is hidden.

If it is a regular column (n>0), then if USETITLESIZE=YES, then it will use the width of the title of the column. Finally if the condition is not true or the width of the title of the column is 0, then the default value WIDTHDEF is used.

RASTERWIDTHn: Same as WIDTHn but in pixels. Has lower priority than WIDTHn. The returned value is the actual computed size.

WIDTHn: Width of column n in SIZE units, where n is the number of the column (n>=0). If the width value is 0, the column will not be shown on the screen. It does not includes the decoration size occupied by the frame lines. The returned value is the actual computed size.

WIDTHDEF: Default column width in SIZE units. Not used for the title column. Default: 80 (width corresponding to 20 characters).

Line Size Attributes

For all lines if HEIGHTn is not defined, then RASTERHEIGHTn is used. If also not defined, then depending on the circumstances a logic is used to find the line height.

If it is the title line (n=0), then if USETITLESIZE=YES or not in callback mode, it will search for the maximum height among the titles of all columns. Finally if the conditions are not true or the maximum height of the line is 0, then the line of column titles is hidden.

If it is a regular line (n>0), then if USETITLESIZE=YES, then it will use the height of the title of the line. Finally if the condition is not true or the height of the title of the line is 0, then the default value HEIGHTDEF is used.

HEIGHTn: Height of line n in SIZE units, where n is the number of the line (n>=0). If the height value is 0, the line will not be shown on the screen. It does not includes the decoration size occupied by the frame lines. The returned value is the actual computed size.

HEIGHTDEF: Default line height in SIZE units. Not used for the title line. Default: 8 (height corresponding to 1 line).

RASTERHEIGHTn: Same as HEIGHTn but in pixels. Has lower priority than HEIGHTn. The returned value is the actual computed size.

Number of Cells Attributes

When lines or columns are added or removed the existing cell, line and column attributes are preserved, except custom application attributes.

ADDCOL (write-only): Adds a new column to the matrix after the specified column. To insert a column at the top of the spreadsheet, value 0 must be used. To add more than one column, use format "***C*-*C***", where the first number corresponds to the base column and the second number corresponds to the number of columns to be added. It can be used in normal operation mode or in callback mode, but in callback mode will not update cell values this must be done by the application. Can NOT add a title column. Ignored if set before map.

ADDLIN (write-only): Adds a new line to the matrix after the specified line. To insert a line at the top of the spreadsheet, value 0 must be used. To add more than one line, use format "***L*-*L***", where the first number corresponds to the base line and the second number corresponds to the number of lines to be added. It can be used in normal operation mode or in callback mode, but in callback mode will not update cell values this must be done by the application. Can NOT add a title line. Ignored if set before map.

DELCOL (write-only): Removes the given column from the matrix. To remove more than one column, use format "***C*-*C***", where the first number corresponds to the base column and the second number corresponds to the number of columns to be removed. It can be used in normal operation mode or in callback mode, but in callback mode will not update cell values this must be done by the application. Can NOT remove a title column, ***C***>0. Ignored if set before map.

DELLIN (write-only): Removes the given line from the matrix. To remove more than one line, use format "***L*-*L***", where the first number corresponds to the base line and the second number corresponds to the number of lines to be removed. It can be used in normal operation mode or in callback mode, but in callback mode will not update cell values this must be done by the application. Can NOT remove a title line, ***L***>0. Ignored if set before map.

NUMCOL: Defines the number of columns in the matrix. Must be an integer number. Default: "0". It does not include the title column. If changed after map will add empty cells or discard cells at the end.

NUMCOL_VISIBLE: When set defines the number of visible columns to be counted when calculating the **Natural** size, not counting the title column. Not used elsewhere. The **Natural** size will always include the title column if any. Can be greater than the actual number of columns, so room will be reserved for adding new columns without the need to resize the matrix. Also it will always use the first columns of the matrix, except if **NUMCOL_VISIBLE_LAST**=YES then it will use the last columns. The remaining columns will be accessible only by using the scrollbar. IMPORTANT: When retrieved returns the current number of visible columns, not including the non scrollable columns. Default: "4".

NUMCOL_NOSCROLL: Number of columns that are non scrollable, not counting the title column. Default: "0". It does not affect the NUMCOL_VISIBLE attribute behavior nor Natural size computation. It will always use the first columns of the matrix. The cells appearance will be the same of ordinary cells, and they can also receive the focus and be edited. Must be less than the total number of columns. (since 3.5)

NUMLIN: Defines the number of lines in the matrix. Must be an integer number. Default: "0". It does not include the title line. If changed after map will add empty cells or discard cells at the end.

NUMLIN_VISIBLE: When set defines the number of visible lines to be counted when calculating the **Natural** size, not counting the title line. Not used elsewhere. The **Natural** size will always include the title line if any. Can be greater than the actual number of lines, so room will be reserved for adding new lines without the need to resize the matrix. Also it will always use the first lines of the matrix, except if **NUMLIN_VISIBLE_LAST**=YES then it will use the last lines. The remaining lines will be accessible only by using the scrollbar. IMPORTANT: When retrieved returns the current number of visible lines, not including the non scrollable lines. Default: "3".

NUMLIN_NOSCROLL: Number of lines that are non scrollable, not counting the title line. Default: "0". It does not affect the NUMLIN_VISIBLE attribute behavior nor Natural size computation. It will always use the first lines of the matrix. The cells appearance will be the same of ordinary cells, and they can also receive the focus and be edited. Must be less than the total number of lines. (since 3.5)

Mark Attributes

MARKAREA: Defines if the area to be **interactively** marked by the user must be continuous or not, valid only if MARKMULTIPLE=YES. Possible values: "CONTINUOUS" or

"NOT_CONTINUOUS". Default: "CONTINUOUS".

MARKATTITLE: a click at a title will mark a full line or a full column if they can be marked. Default: "Yes". (since 3.16)

MARKMODE: Defines the entity that can be marked: none, lines, columns, (lines or columns), and cells. Possible values: "NO", "LIN", "COL", "LINCOL" or "CELL". Default: "NO" (no mark).

MARK*L:C* (no redraw): marks a cell, a line or a column depending on MARKMODE, and returns cell, line or column mark state also according to MARKMODE. Can be "1" or "0". If MARKMODE=LIN,COL,LINCOL use 0 to mark only the other element (ex: "0:3" set/get for column 3). Even when MARKMODE=LIN,COL,LINCOL you can specify a single cell address. (since 3.0)

MARKED: String of '0' or '1' characters, informing which cells are marked (indicated by value '1'). Use NULL to clear all marks, returns NULL if no marks. The format of this character vector depends on the value of the MARKMODE attribute: if its value is CELL, the vector will have NUMLIN x NUMCOL positions, corresponding to all the cells in the matrix starting with all the cells of the first line, then the second line and so on. If its value is LIN, the vector will begin with letter 'L' and will have further NUMLIN positions, each one corresponding to a line in the matrix. If its value is COL, the vector will begin with letter 'C' and will have further NUMCOL positions, each one corresponding to a column in the matrix. If its value is LINCOL, the first letter, which can be either 'L' or 'C', will indicate which of the above formats is being used. If you change the other mark attributes the marked cells are cleared. When setting the attribute the LIN and COL notation can be used even if MARKMODE=CELL. MULTIPLE and AREA are NOT considered when setting MARKED or MARKL:C.

MARKMULTIPLE: Defines if more than one entity defined by MARKMODE can be **interactively** marked. Possible values: "YES" or "NO". Default: "NO".

Action Attributes

CLEARATTRIB (write-only): Clear all cell attributes if ALL, all attributes except titles if CONTENTS, and all selected cell attributes if MARKED. When ALL is specified, all lines and column attributes are also cleared. (since 3.6)
CLEARATTRIB*L:C* (write-only): Clear all cell attributes in an interval starting at the specified cell. Its value defines the end cell in the "L:C" format, the default is the last cell. (since 3.6)
CLEARATTRIB*L:** (write-only): the cell attributes in line L. Its value defines a column inclusive interval in the "C1-C2" format. The default is 0 and the last column. When a full line is specified, all line attributes are also cleared. (since 3.6)
CLEARATTRIB*:C* *(write-only)*: the cell attributes in column C. Its value defines a line inclusive interval in the "L1-L2" format. The default is 0 and the last line. When a full column is specified, all column attributes are also cleared, including ALIGNMENT and SORTSIGN. (since 3.6)

In all cases, attributes are set to NULL. Only the attributes FONT*, BGCOLOR*, FGCOLOR*, FRAMEHORIZCOLOR*, FRAMEHORIZCOLOR*, ALIGNMENT* and SORTSIGN* are affected. In callback mode will not call the user callbacks.

CLEARVALUE (write-only): Clear all values if ALL, all values except titles if CONTENTS, and all selected cell values if MARKED. (since 3.6)
CLEARVALUE*L:C* (write-only): Clear all values in an interval starting at the specified cell. Its value defines the end cell in the "L:C" format, the default is the last cell. (since 3.6)
CLEARVALUE*L:** (write-only): the values in line L. Its value defines a column inclusive interval in the "C1-C2" format. The default is 0 and the last column. (since 3.6)
CLEARVALUE*:C* (write-only): the values in column C. Its value defines a line inclusive interval in the "L1-L2" format. The default is 0 and the last line. (since 3.6)

In all cases, values are set to NULL. Works also in callback mode.

COPYCOL*C* (write-only): copy the values and attributes from column C to the given column (value is the number of a column). (Since 3.9)

COPYLIN*L* (write-only): copy the values and attributes from line L to the given line (value is the number of a line). (Since 3.9)

FITTOSIZE (write-only): Force lines and/or columns sizes so the matrix visible size fit in its current size. NUMCOL_VISIBLE and NUMLIN_VISIBLE are considered when fitting and they are not changed, only the RASTERWIDTHn and RASTERHEIGHTn attributes are changed. But if any of the RASTERWIDTHn and RASTERHEIGHTn attributes where already set, then they will not be changed. If the matrix is resized then it must be set again to obtain the same result, but before doing that set to NULL all the RASTERWIDTHn and RASTERHEIGHTn attributes that you want to be changed. Can be LINES, COLUMNS or YES (meaning both). (since 3.3)

FITTOTEXT (write-only): Fit the RASTERWIDTHn or the RASTERHEIGHTn attribute for the given column or line, so that it will fit the largest text in the column or the highest text in the line. The number of the column or line must be preceded by a character identifying its type, "C" for columns and "L" for lines. For example "C5"=column 5 or "L3"=line 3. If FITMAXWIDTHn or FITMAXHEIGHTn are set for the column or line they are used as maximum limit for the size. (since 3.4)

MOVECOL*C* (write-only): move the values and attributes from column C to the given column (value is the number of a column). Internally will use ADDCOL+COPYCOL+DELCOL to perform the move so it is limited to those attributes restrictions. It can be used in normal operation mode or in callback mode, but in callback mode will not update cell values, this must be done by the application. (Since 3.9)

MOVELIN*L* (write-only): move the values and attributes from line L to the given line (value is the number of a line). Internally will use ADDLIN+COPYLIN+DELLIN to perform the move so it is limited to those attributes restrictions. It can be used in normal operation mode or in callback mode, but in callback mode will not update cell values, this must be done by the application. (Since 3.9)

REDRAW (write-only): The user can inform the matrix that the data has changed, and it must be redrawn. Values:

"ALL": Redraws the whole matrix.
"L%d": Redraws the given line (e. g.: "L3" redraws line 3)
"L%d-%d": Redraws the lines in the given region (e.g.: "L2-4" redraws lines 2, 3 and 4)
"C%d": Redraws the given column (e.g.: "C3" redraws column 3)
"C%d-%d": Redraws the columns in the given region (e.g: "C2-4" redraws columns 2, 3 and 4)

No redraw is done when the application sets the attributes: L:C, ALIGNMENTc, BGCOLOR*, FGCOLOR*, FONT*, VALUE, FRAME*COLOR, MARKL:C. Global and size attributes always automatically redraw the matrix.

SHOW (write-only): If necessary scroll the visible area to make the given cell visible. To scroll to a line or a column, use a value such as "***L***:*" or "*:***C***" (where ***L***>0 and ***C***>0). (since 3.0)

Editing Attributes

EDITMODE: When set to YES, programmatically puts the current cell in edition mode, allowing the user to modify its value. When consulted informs if the editing control is visible (text or dropdown). Possible values: "YES" or "NO".

EDITALIGN: sets the text box alignment to the column alignment when editing a cell value. Default: No. (since 3.14)

EDITCELL (read-only): returns the current cell being edited ("L:C"), or NULL if none. Can also be used during interaction while editing is being performed and EDITHIDEONFOCUS=NO. (since 3.14)

EDITFITVALUE: enable a text box larger than the cell size of necessary, according to the cell font and cell current value. While editing if more room is necessary it will grow to the right. (since 3.14)

EDITHIDEONFOCUS: when editing a cell if text box loses its focus, then editing ends. Default: Yes. When set to NO editing will continue and the matrix can be scrolled, also when pressing Esc or Enter if the focus is at the matrix it has the same effect as if pressed at the text box. (since 3.14)

EDITING (read-only): returns Yes if the editing process is active for text or dropdown. It is set to Yes after EDITION_CB, after MENUDROP_CB, before DROP_CB and before the editing control is made visible. Set to NO when editing is about to end, after EDITION_CB and after the value has been updated, but before the editing control is made invisible. (since 3.13)

EDITNEXT: controls how the next cell after editing is chosen. Can be LIN, COL, LINCR, COLCR. Default: LIN. (since 3.4)

LIN - go to the next line, if at last line then go to the next column at the same line;
LINCR - go to the next line, if at last line then go to the next column at the first line;
COL - go to the next column, if at last column then go to the next line at the same column;
COLCR - go to the next column, if at last column then go to the next line at the first column;
NONE - stay in the same cell. (since 3.6)

EDITTEXT (read-only): returns Yes if the editing is being done by a text box. (since 3.14)

EDITVALUE (read-only): returns Yes if the display cell value being consulted will be used for a text box initial value. Useful for being consulted inside the translate and numeric callbacks. (since 3.14)

Text Editing Attributes

CARET: Allows specifying and verifying the caret position of the text box in edition mode.

INSERT: inserts a text at the caret position of the text box in edition mode. (since 3.14)

MASK*L:C* or **MASK*L:**** or **MASK**:C***: Defines a mask that will filter text input. All MASK auxiliary attributes are also available by adding the line and column at the end of the attribute name. (lin and col * variations since 3.17)

MULTILINE: allows the edition of multiple lines. Use Shift+Enter to add lines. Enter will end the editing.

SELECTION: Allows specifying and verifying selection interval of the text box in edition mode.

Canvas Attributes (inheritable)

BORDER: Changed to NO.

SCROLLBAR: Changed to YES.

IupMatrix Callbacks

Interaction

ACTION_CB: Action generated when a keyboard event occurs.

```
int function(Ihandle *ih, int key, int lin, int col, int edition, char* value); [in C]
ih:action_cb(key, lin, col, edition: number, value: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
key: Identifier of the typed key. Please refer to the Keyboard Codes table for a list of possible values.
lin, **col**: Coordinates of the selected cell.
edition: 1 if the cell is in edition mode, and 0 if it is not.
value: When EDITMODE=NO is the cell current value, but if the type key is a valid character then contains a string with that character. When EDITMODE=Yes depends on the editing field type. If a dropdown, then it is an empty string (""). If a text, and the type key is a valid character then it is the future value of the text field, if not a valid character then it is the cell current value. Notice that this value can be NULL if the cell does not have a value and the key pressed is not a character.

Returns: IUP_DEFAULT validates the key, IUP_IGNORE ignores the key, IUP_CONTINUE forwards the key to IUPs conventional processing, or the identifier of the key to be treated by the matrix.

CLICK_CB: Action generated when any mouse button is pressed over a cell. This callback is always called after other callbacks. When EDITHIDEONFOCUS=NO and editing is on going the callback EDITCLICK_CB with the same parameters will also be called right before this one (since 3.14).

```
int function(Ihandle *ih, int lin, int col, char *status); [in C]
ih:click_cb(lin, col: number, status: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell where the mouse button was pressed.
status: Status of the mouse buttons and some keyboard keys at the moment the event is generated. The same macros used for BUTTON_CB can be used for this status.

Returns: To avoid the display update return IUP_IGNORE.

COLRESIZE_CB: Action generated when a column is interactively resized. (Since 3.9)

```
int function(Ihandle *ih, int col); [in C]
ih:colresize_cb(col: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
col: Column that had its size changed.

RELEASE_CB: Action generated when any mouse button is released over a cell. This callback is always called after other callbacks. When EDITHIDEONFOCUS=NO and editing is on going the callback EDITRELEASE_CB with the same parameters will also be called right before this one (since 3.14).

```
int function(Ihandle *ih, int lin, int col, char *status); [in C]
ih:release_cb(lin, col: number, status: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell where the mouse button was pressed.
status: Status of the mouse buttons and some keyboard keys at the moment the event is generated. The same macros used for BUTTON_CB can be used for this status.

Returns: To avoid the display update return IUP_IGNORE.

RESIZEMATRIX_CB: Action generated after the element size has been updated but before the cells have been actually refreshed. (Since 3.10.1)

```
int function(Ihandle *ih, int width, int height); [in C]
ih:resizematrix_cb(width, height: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
width: the width of the internal element size in pixels not considering the BORDER size (client size)
height: the height of the internal element size in pixels not considering the BORDER size (client size)

MOUSEMOVE_CB: Action generated to notify the application that the mouse has moved over the matrix. When EDITHIDEONFOCUS=NO and editing is on going the callback EDITMOUSEMOVE_CB with the same parameters will also be called right before this one (since 3.14).

```
int function(Ihandle *ih, int lin, int col); [in C]
ih:mousemove_cb(lin, col: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell that the mouse cursor is currently on.

ENTERITEM_CB: Action generated when a matrix cell is selected, becoming the current cell. Also called when matrix is getting focus. Also called when focus is changed because lines or columns were added or removed (since 3.9).

```
int function(Ihandle *ih, int lin, int col); [in C]
ih:enteritem_cb(lin, col: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the selected cell.

LEAVEITEM_CB: Action generated when a cell is no longer the current cell. Also called when the matrix is losing focus.

```
int function(Ihandle *ih, int lin, int col); [in C]
ih:leaveitem_cb(lin, col: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell which is no longer the current cell.

Returns: IUP_IGNORE prevents the current cell from changing, but this will not work when the matrix is losing focus. If you try to move to beyond matrix borders the cell will lose focus and then get it again, so leaveitem_cb and enteritem_cb will be called.

SCROLLTOP_CB: Action generated when the matrix is scrolled with the scrollbars or with the keyboard. Can be used together with the ORIGIN and ORIGINOFFSET attributes to synchronize the movement of two or more matrices.

```
int function(Ihandle *ih, int lin, int col); [in C]
ih:scrolltop_cb(lin, col: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell currently in the upper left corner of the matrix.

Drawing

BGCOLOR_CB - Action generated to retrieve the background color of a cell when it needs to be redrawn.

```
int function(Ihandle *ih, int lin, int col, int *red, int *green, int *blue); [in C]
ih:bgcolor_cb(lin, col: number) -> (red, green, blue, ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell.
red, **green**, **blue**: the cell background color.

Returns: If IUP_IGNORE, the values are ignored and the attribute defined background color will be used. If returns IUP_DEFAULT the returned values will be used as the background color.

FGCOLOR_CB - Action generated to retrieve the foreground color of a cell when it needs to be redrawn.

```
int function(Ihandle *ih, int lin, int col, int *red, int *green, int *blue); [in C]
ih:fgcolor_cb(lin, col: number) -> (red, green, blue, ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell.
red, green, blue: the cell foreground color.

Returns: If IUP_IGNORE, the values are ignored and the attribute defined foreground color will be used. If returns IUP_DEFAULT the returned values will be used as the foreground color.

FONT_CB: Action generated to retrieve the font of a cell. Called both for common cells and for line and column titles. (since 3.0)

```
char* function(Ihandle* ih, int lin, int col); [in C]
ih:font_cb(lin, col: number) -> (ret: string) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell.

Returns: Must return a font or NULL to use the the attribute defined font.

TYPE_CB: Action generated to retrieve the type of a cell value. Called both for common cells and for line and column titles. (since 3.9)

```
char* function(Ihandle* ih, int lin, int col); [in C]
ih:type_cb(lin, col: number) -> (ret: string) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell.

Returns: Must return "TEXT", "COLOR", "FILL" or "IMAGE".

DRAW_CB: Action generated before a cell is drawn. Allows to draw a custom cell contents. You must use the CD library primitives. The clipping is set for the bounding rectangle. The callback is called after the cell background has been filled with the background color. The focus feedback area is not included in the decoration size.

```
int function(Ihandle *ih, int lin, int col, int x1, int x2, int y1, int y2, cdCanvas* cnv); [in C]
ih:draw_cb(lin, col, x1, x2, y1, y2: number, cnv: cdCanvas) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the current cell.
x1, **x2**, **y1**, **y2**: Bounding rectangle of the current cell in pixels, excluding the decorations.
cnv: internal canvas CD used to draw the matrix.

Returns: If IUP_IGNORE the normal text drawing will take place.

DROPCHECK_CB: Action generated before the current cell is redrawn to determine if a dropdown/popup menu feedback or a toggle should be shown. If this action is not registered, no feedback will be shown. If the callback is defined and return IUP_DEFAULT for a cell, to show the dropdown/popup menu the user can simply do a single click in the drop feedback area of that cell. (Toggle support since 3.9)

```
int function(Ihandle *ih, int lin, int col); [in C]
ih:dropcheck_cb(lin, col: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell.

Returns: IUP_DEFAULT will show a drop feedback, IUP_CONTINUE will show and enable the toggle button, or IUP_IGNORE to draw nothing.

TRANSLATEVALUE_CB: Action generated to translate the value of a cell during display and size computation. Called both for common cells and for line and column titles. (since 3.13)

```
char* function(Ihandle* ih, int lin, int col, char* value); [in C]
ih:translatevalue_cb(lin, col: number, value: string) -> (ret: string) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell.
value: original cell value

Returns: the string to be drawn.

Editing (not called if READONLY=Yes)

TOGGLEVALUE_CB: Action generated when a toggle button is pressed. (Since 3.9)

```
int function(Ihandle *ih, int lin, int col, int status); [in C]
ih:togglevalue_cb(lin, col, status: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell where the mouse button was pressed.
status: Value of the toggle. Can be 1 or 0.

VALUECHANGED_CB: Called after the value was interactively changed by the user or after a group of values where programmatically changed in a single operation (since 3.9). When it was interactively changed the temporary attribute CELL_EDITED will be set to Yes during the callback (since 3.13).

```
int function(Ihandle *ih); [in C]
ih:valuechanged_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

DROP_CB: Action generated before the current cell enters edition mode to determine if a text field or a dropdown list will be shown. It is called after EDITION_CB. If this action is not registered, a text field will be shown. Its return determines what type of element will be used in the edition mode. If the selected type is a dropdown, the values appearing in the dropdown must be fulfilled in this callback, just like elements are added to any list (the drop parameter is the handle of the dropdown list to be shown). You should also set the lists current value ("VALUE"), the default is always "1". The previously cell value can be verified from the given drop Ihandle via the "PREVIOUSVALUE" attribute.

```
int function(Ihandle *ih, Ihandle *drop, int lin, int col); [in C]
ih:drop_cb(drop: ihandle, lin, col: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
drop: Identifier of the dropdown list which will be shown to the user.
lin, **col**: Coordinates of the current cell.

Returns: IUP_IGNORE to show a text-edition field, or IUP_DEFAULT to show a dropdown field.

MENUDROP_CB: Action generated before the current cell enters edition mode to determine if a popup menu will be shown instead of a text field or a dropdown. If this action is registered and retunr IUP_DEFAULT the DROP_CB callback is not called, and the popup menu is shown. Like DROP_CB, it is called after EDITION_CB. The values appearing as menu items in the popup menu must be fulfilled in this callback, like elements are added to a list (the drop parameter is the handle of the popup menu to be shown, but the actual items will be added later by the internal processing). You could also set the "VALUE" attribute that will add a mark to the menu item with the same number. If IMAGEid is set then an IMAGE attribute will be set at the correspondent menu item. The previously cell value can be verified from the given drop Ihandle via the "PREVIOUSVALUE" attribute. (since 3.6)

```
int function(Ihandle *ih, Ihandle *drop, int lin, int col); [in C]
ih:drop_cb(drop: ihandle, lin, col: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
drop: Identifier of the popup menu which will be shown to the user.
lin, **col**: Coordinates of the current cell.

Returns: IUP_IGNORE to not show the menu for the given cell, DROP_CB will then be called.

DROPSELECT_CB: Action generated when an element in the dropdown list or the popup menu is selected. For the dropdown, if returns IUP_CONTINUE the value is accepted as a new value and the matrix leaves edition mode, else the item is selected and editing remains. For the popup menu the returned value is ignored.

```
int function(Ihandle *ih, int lin, int col, Ihandle *drop, char *t, int i, int v); [in C]
ih:dropselect_cb(lin, col: number, drop: ihandle, t: string, i, v: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the current cell.
drop: Identifier of the dropdown list or the popup menu shown to the user.
t: Text of the item whose state was changed.
i: Number of the item whose state was changed.
v: Indicates if item was selected or unselected (1 or 0). Always 1 for the popup menu.

EDITION_CB: Action generated when the current cell enters or leaves the edition mode. Not called if READONLY=YES.

```
int function(Ihandle *ih, int lin, int col, int mode, int update);  [in C]
ih:edition_cb(lin, col, mode, update: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the current cell.
mode: 1 if the cell has entered the edition mode, or 0 if the cell has left the edition mode.
update: used when mode=0 to identify if the value will be updated when the callback returns with IUP_DEFAULT. (since 3.0)

Returns: can be IUP_DEFAULT, IUP_IGNORE or IUP_CONTINUE.

If the callback does not exists the cell can always be edited and the new value is always accepted.

When editing is started, **mode**=1 and **update**=0. Editing is allowed if the callback returns IUP_DEFAULT, so to make the cell read-only return IUP_IGNORE.

When editing ends, **mode**=0 and **update** can be 0 or 1. The new value is accepted only if the callback returns IUP_DEFAULT. The VALUE attribute when consulted inside the callback returns the new value that will be updated to the cell. **update**=0 only when the user cancel the editing by pressing the **Esc** key. If the callback returns IUP_CONTINUE the edit mode is ended and the new value will not be updated, so the application can set a different value during the callback (useful to format the new value). If the callback returns IUP_IGNORE the editing is not ended, with several exceptions: the **Esc** key was used; the matrix size, scroll or visibility was changed during edition mode; a click in another cell; or the edit control loses its focus.

This callback is also called when the user press **Del** to clear the cell contents or other multiple cell editing. The callback will simply validate the operation for each cell been cleared by checking if the matrix is read-only or if the cell is read-only. In this situation it is called with **mode**=1 and **update**=1. When in normal mode (not callback mode) the new value can not be refused, but you can use the VALUE_EDIT_CB to reset a new value or use the VALUECHANGED_CB to check all the new values after they where changed.

Callback Mode

VALUE_CB: Action generated to retrieve the value of a cell. Called both for common cells and for line and column titles.

```
char* function(Ihandle* ih, int lin, int col); [in C]
ih:value_cb(lin, col: number) -> (ret: string) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell.

Returns: the string to be drawn.

IMPORTANT: The existence of this callback defines the callback operation mode of the matrix when it is mapped.

VALUE_EDIT_CB: Action generated to notify the application that the value of a cell was changed. Never called when READONLY=YES. This callback is usually set in callback mode, but also works in normal mode. When in normal mode, it is called after the new value has been internally stored, so to refuse the new value simply reset the cell to the desired value. When it was interactively changed the temporary attribute CELL_EDITED will be set to Yes during the callback (since 3.13).

```
int function(Ihandle *ih, int lin, int col, char* newval); [in C]
ih:value_edit_cb(lin, col: number, newval: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell.
newval: String containing the new cell value

IMPORTANT: if VALUE_CB is defined and VALUE_EDIT_CB is not defined when the matrix is mapped it will be read-only.

MARK_CB: Action generated to retrieve the selection state of a cell. Called only for common cells, only when MARKMODE=CELL and only in callback mode.

```
int function(Ihandle* ih, int lin, int col); [in C]
ih:mark_cb(lin, col: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell.

Returns: the selection state (marked=1, not marked 0). If not defined the attribute "**MARK*L:C***" will be returned.

MARKEDIT_CB: Action generated to notify the application that the selection state of a cell was changed. Since it is a notification, it cannot refuse the mark modification. Called only for common cells, only when MARKMODE=CELL and only in callback mode.

```
int function(Ihandle *ih, int lin, int col, int marked); [in C]
ih:markedit_cb(lin, col, marked: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, **col**: Coordinates of the cell.
marked: selection state (marked=1, not marked 0).

If not defined the attribute "**MARK*L:C***" will be updated. So if you define the **MARKEDIT_CB** the "**MARK*L:C***" will NOT be updated and the callback **MARK_CB** must return the selection state of the cell. If you do not want to implement the **MARK_CB** callback then set the "**MARK*L:C***" attribute inside the **MARKEDIT_CB** callback.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

The **IupCanvas** callbacks ACTION, SCROLL_CB, KEYPRESS_CB, MOTION_CB, FOCUS_CB, RESIZE_CB and BUTTON_CB can be changed but you should save and call the original definition from inside your own callback, or the matrix will not correctly work. This can **not** be done in Lua, except for BUTTON_CB, MOTION_CB and KEYPRESS_CB that are exported to Lua as "**MatButtonCb**", "**MatMotionCb**" and "**MatKeyPressCb**" functions.

Use IupConvertXYToPos to convert (x,y) coordinates in the cell position, then use IupTextConvertPosToLinCol to convert pos into (lin,col), or use the formula "pos=lin*(NUMCOL+1) + col". Here lin and col starts at 0, pos starts at 0.

See IupCanvas.

Returns the identifier of the created matrix, or NULL if an error occurs.

Attributes

"1": First item in the list.
"2": Second item in the list.
"3": Third item in the list.
...
"id": idth item in the list.

(non inheritable) Item value. It can be any text. Differently from the **IupList** control, the item must exist so its label can be changed. So **APPENDITEM, INSERTITEM*id*, ADDLIN** or **COUNT** attributes must be used to reserve space for the list items. Notice that lines and items in the list are the same thing.

ADDLIN (write-only): adds a new line to the list after the specified line. To insert a line at the top, value 0 must be used. To add more than one line, use format "***L*-*L***", where the first number corresponds to the base line and the second number corresponds to the number of lines to be added. Ignored if set before map.

APPENDITEM (write-only): inserts an item after the last item. Ignored if set before map.

COLORCOL (read-only): returns the number of color column. If not exists, returns 0.

COLOR*id*: the color displayed at the color column. If not defined the color box is not diplayed.

COLUMNORDER: defines or retrieves the display order of the columns. Possible values a combination of: "LABEL", "COLOR" and "IMAGE". These values also can be combined in these formats: VALUE1 (one column); VALUE1:VALUE2 (two columns) or VALUE1:VALUE2:VALUE3 (three columns). Default: "LABEL" (one column).

COUNT: defines the number of items in the list. Differently from the **IupList** control it is not read-only. It does not include the extra empty item when EDITABLE=Yes.

DELLIN (write-only): removes the given line from the list. To remove more than one line, use format "***L*-*L***", where the first number corresponds to the base line and the second number corresponds to the number of lines to be removed. Ignored if set before map.

EDITABLE (creation-only): enables the interactive editing of the list. It can be Yes or No. Default: "NO". An empty item at the end of the list will be available so new items can be interactively inserted. Also while editing a label, the IMAGE column will display a button so the item can be interactively removed.

FOCUSCOLOR: the background color when an item get the focus. Values in RGB format ("r g b"). Default: "255 235 155".

FOCUSITEM: defines the current focus item.Default: "1".

IMAGE*id* (write-only): name of the image to be used in the specified item (id). Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. Image column must be available.

IMAGEACTIVE*id*: controls the interaction with the image of an item. It can be Yes or No. Default: Yes. Image column must be available.

IMAGEADD (write-only): name of the image that will be shown when the IupMatrixList is editable. Default: "MTXLIST_IMG_ADD". Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. Image column must be available.

IMAGECHECK (write-only): name of the image that will be shown when the IMAGEVALUE attribute is "IMAGECHECK". Default: "MTXLIST_IMG_CHECK". Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. Image column must be available.

IMAGECOL (read-only): returns the number of image column. If not exists, returns 0.

IMAGEDEL (write-only): name of the image that will be shown when the IupMatrixList is editable or when SHOWDELETE=Yes. Default: "MTXLIST_IMG_DEL". Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. Image column must be available.

IMAGEUNCHECK (write-only): name of the image that will be shown when the IMAGEVALUE attribute is "IMAGEUNCHECK". Default: "MTXLIST_IMG_UNCHECK". Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. Image column must be available.

IMAGEVALUE*id*: selects the CHECK or the UNCHECK image to display for an item (id). It can be Yes or No. Default: NO.

INSERTITEM*id* (write-only): inserts an item before the given id position (id starts at 1). If id=COUNT+1 then it will append after the last item. Ignored if out of bounds. Ignored if set before map.

ITEMACTIVE*id*: controls the interaction with an item (id). It can be Yes or No. Default: "YES".

ITEMFGCOLOR*id*: text color of an item (id).

ITEMBGCOLOR*id*: background color of an item (id).

LABELCOL (read-only): returns the number of label column. If not exists, returns 0.

REMOVEITEM (write-only): removes the given item from the list.

SHOWDELETE: Shows only the **IMAGEDEL** image and ignores **IMAGECHECK** and **IMAGEUNCHECK.**

TITLE: title of the list. When not NULL the list will display a non scrollable title.

TOPITEM (write-only): position the given item at the top of the list or near to make it visible.

VALUE: defines or retrieves the value of the current cell.

VISIBLELINES: defines the number of visible lines for the **Natural Size**, this means that will act also as minimum number of visible lines. Default: "3".

Other Attributes

Since the **IupMatrixList** inherits its implementation from the **IupMatrix**, and that one from **IupCanvas**, those controls attributes and callbacks can be used. But notice that **IupMatrixList** uses several of them internally for its own purpose, and reusing them may affect the control behavior and appearance.

Some attribute defaults were changed:

EXPAND: changed to "NO".

ALIGNMENTLIN0: changed to "ALEFT".

CURSOR: changed to "ARROW".

FRAMETITLEHIGHLIGHT: changed to "NO".

HIDEFOCUS: changed to "YES".

SCROLLBAR: changed to "VERTICAL".

ACTIVE, EXPAND, FONT, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

IMAGEVALUECHANGED_CB: called after the image value was interactively changed by the user (mark/unmark).

```
int function (Ihandle *ih, int lin, int imagevalue); [in C]
ih:imagevaluechanged_cb(lin, imagevalue: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin: item line.
imagevalue: equal to 1 if the image used was IMAGECHECK or to 0 if the image used IMAGEUNCHECK.

LISTACTION_CB: Action generated when the state of an item in the list is changed. Also provides information on the changed item:

```
int function (Ihandle *ih, int item, int state); [in C]
ih:listaction_cb(item, state: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
item: Number of the changed item starting at 1.
state: Equal to 1 if the item is in focus or to 0 if the item loses its focus.

LISTCLICK_CB: Action generated when any mouse button is pressed over a item.

```
int function (Ihandle *ih, int lin, int col, char *status); [in C]
ih:listclick_cb(lin, col: number, status: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin: item line.
col: item column (label, image or color).
status: Status of the mouse buttons and some keyboard keys at the moment the event is generated. The same macros used for BUTTON_CB can be used for this status.

Returns: To avoid the default processing return IUP_IGNORE.

LISTDRAW_CB: Action generated when an item needs to be redrawn. It is called before the default processing.

```
int function (Ihandle *ih, int lin, int col, int x1, int x2, int y1, int y2, cdCanvas* cnv); [in C]
ih:listdraw_cb(text: string, item, state: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin: item line.
col: item column (label, image or color).
x1, **x2**, **y1**, **y2**: bounding rectangle of the current cell in pixels, excluding the decorations.
cnv: internal canvas CD used to draw the list.

Returns: If IUP_IGNORE the normal drawing will take place.

LISTEDITION_CB: Action generated when the current cell of an item enters or leaves the edition mode. Called before the default processing.

```
int function (Ihandle *ih, int lin, int col, int mode, int update); [in C]
ih:listedition_cb(lin, col, mode, update: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin: item line.
col: item column (label, image or color).
mode: equal to 1 if the cell has entered the edition mode, or 0 if the cell has left the edition mode.
update: equal to 1 to redraw, or 0 to no update returning IUP_IGNORE.

LISTINSERT_CB: Action generated when a new item is inserted into the list.

```
int function (Ihandle *ih, int lin); [in C]
ih:listinsert_cb(lin: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin: position of the new item.

LISTRELEASE_CB: Action generated when any mouse button is released over a item.

```
int function (Ihandle *ih, int lin, int col, char *status); [in C]
ih:listrelease_cb(lin, col: number, status: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin: item line.
col: item column (label, image or color).
status: Status of the mouse buttons and some keyboard keys at the moment the event is generated. The same macros used for BUTTON_CB can be used for this status.

Returns: To avoid the default processing return IUP_IGNORE.

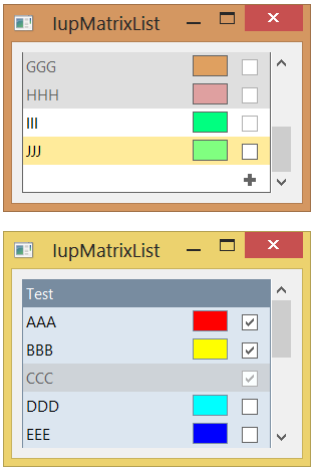
LISTREMOVE_CB: Action generated when an item is removed of the list.

```
int function (Ihandle *ih, int lin); [in C]
ih:listremove_cb(lin: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin: position of the removed item.

Examples

Browse for Example Files



See Also

IupCanvas, IupMatrix

Shortcut Keys

The library adds some shortcut keys to the already implemented in **IupMatrix**:

```
Ctrl+A (English)    => Select All
Ctrl+T (Portuguese) => Select All
Ctrl+X              => Cut (Copy + Clear Cell Values)
Ctrl+C              => Copy to Clipboard (marked cells)
Ctrl+V              => Paste from Clipboard (start at focus cell)
Ctrl+Z              => Undo 1 level
Ctrl+Y (English)    => Redo 1 level
Ctrl+R (Portuguese) => Redo 1 level
Ctrl+F (English)    => Show the Find Dialog
Ctrl+L (Portuguese) => Show the Find Dialog
Alt+F3              => Show the Find Dialog
Esc                 => Hide the Find Dialog
F3                  => Find Next
Shift+F3            => Find Previous
Ctrl+G              => Show the Go To Cell Dialog
```

Available Quantity and Units

Unit names, symbols and conversion factors were almost all based on:

http://en.wikipedia.org/wiki/Conversion_of_units

By definition, unit names and symbols follow the case displayed in the table. When setting the NUMERICQUANTITY and NUMERICUNIT attributes use English names, the case is insensitive and spaces are ignored. Some Quantities have alternative names, once used the returned values in the attribute will be the same alternative name. For example, you can use "Specific Weight" or "SPECIFICWEIGHT", and you can use "Speed" or "Velocity".

All numeric attributes can be set without the element being mapped to the native system, so the **IupMatrixEx** element can also be used as a Quantity Units database.

The unit used as a reference for conversion is always the first unit listed, and it is the unit defined by the International System of Units (SI). The American spell can be used setting **NUMERICUNITSPELL=AMERICAN.**

NOTICE: These are only a small set of commonly used units. If you need other units, please let us know so we can include them.

Obs: "g" in Comments is the standard gravity. All Quantity and Unit names are described in English. The symbols that have extended characters will work in ISO8859-1 and in UTF-8, according to the UTF8MODE global attribute. The cell background colors are just for clarity and do not imply in any standard classification.

Quantity NUMERICQUANTITY	Units NUMERICUNIT	Symbol NUMERICUNITSYMBOL	Comments
Time	second minute hour day week millisecond microsecond	s min h d wk ms µs	
Mass	kilogram gram tonne pound ounce	kg g t lb oz	- (CGS Unit) - metric ton - (international avoirdupois) - oz = lb / 16
Temperature	Kelvin degree Celsius degree Fahrenheit degree Rankine	K °C °F °Ra	
Length	metre millimetre centimetre kilometre nanometre angstrom micron inch foot yard mile nautical mile	m mm cm km nm Å µ in ft yd mi NM	- (CGS Unit) - micrometre - in = 25.4 mm (international) - ft = 12 in (international) - yd = 3 ft (international) - mi = 1760 yd (international) - NM = 6080 ft (Admiralty)
Area	square metre square centimetre square millimetre square inch square foot square kilometre square yard square mile acre hectare	m² cm² mm² sq in sq ft km² sq yd sq mi ac ha	- (CGS Unit) - ac = 4840 sq yd
Volume	cubic metre cubic centimetre cubic millimetre cubic kilometre cubic inch cubic foot cubic mile cubic yard litre gallon barrel	m³ cm³ mm³ km³ cu in cu ft cu mi cu yd L gal bl	- (CGS Unit) - gal = 231 cu in (US fluid; Wine) - bl = 42 gal (petroleum)
Angle	radian degree gradian	rad ° grad	
Speed (or Velocity)	metre per second inch per second foot per second kilometre per hour centimetre per second mile per hour knot	m/s in/s ft/s km/h cm/s mph kn	 - (CGS Unit) - kn = NM/h
Angular Speed (or Angular Frequency)	radian per second radian per minute degree per second degree per minute Hertz revolution per minute	rad/s rad/min deg/s deg/min Hz rpm	 - revolution per second (frequency)
Acceleration	metre per second squared inch per second squared knot per second mile per second squared standard gravity	m/s² in/s² kn/s mi/s² g	
Kinematic Viscosity	square metre per second square foot per second stokes	m²/s ft²/s St	 - (CGS Unit)
Dynamic Viscosity	pascal second poise pound per foot hour pound per foot second	Pa·s P lb/(ft·h) lb/(ft·s)	
Flow	cubic metre per second cubic inch per second cubic foot per second	m³/s in³/s ft³/s	
Force	Newton Kilonewton dyne kilogram-force pound-force kip-force ton-force	N kN dyn kgf lbf kip tnf	= kg·m/s² = g·cm/s² (CGS Unit) - lbf = g · lb - kip = g · 1000 lb - tnf = g · 2000 lb
Pressure (or Mechanical Stress)	Pascal kilopascal atmosphere millimetre of mercury bar torr pound per square inch kip per square inch	Pa kPa atm mmHg bar torr psi ksi	- Pa = N/m² = kg/(m·s²) - (standard) = mmHg =13595.1 kg/m³·mm·g - psi = lbf/sq in - ksi = kip/sq in

Force per length (or Linear Weight)	Newton per metre Kilonewton per metre kilogram-force per metre ton-force per metre	N/m kN/m kgf/m tnf/m	= kg/s²
Torque (or Moment of Force)	Newton metre kilogram-force metre ton-force metre Newton centimetre kilogram-force centimetre ton-force centimetre Kilonewton-metre metre kilogram	N·m kgf·m tnf·m N·cm kgf·cm tnf·cm kN·m m·kg	= kg·m²/s²
Specific Mass (or Density)	kilogram per cubic metre gram per cubic centimetre gram per millilitre kilogram per litre pound per cubic foot pound per cubic inch pound per gallon	kg/m³ g/cm³ g/mL kg/L lb/ft³ lb/in³ lb/gal	
Specific Weight	Newton per cubic metre Kilonewton per cubic metre kilogram-force per cubic metre ton-force per cubic metre kilogram-force per litre pound-force per cubic foot	N/m³ kN/m³ kgf/m³ tnf/m³ kgf/L lbf/ft³	
Energy	Joule Kilojoule calorie kilocalorie British Thermal Unit Kilowatt-hour horsepower-hour	J kJ cal kcal BTU kW.h hp.h	= m·N = kg·m²/s² - (International Table) - (International Table)
Power (or Heat Flow Rate)	Watt Kilowatt calorie per second horsepower	W kW cal/s hp	= J/s = N·m/s = kg·m²/s³ - (International Table) - hp = 550 ft lbf/s (imperial mechanical)
Electric Charge	Coulomb Faraday milliampere hour	C F mA·h	= A·s
Illuminance	lux footcandle lumen per square inch phot	lx fc lm/in² ph	- lm/m² - lumen per square foot
Fraction	percentage per one per ten per thousand	% /1 /10 /1000	
None			Use numeric values but without using units.

Examples

Browse for Example Files

See Also

IupMatrix

action: Name of the action generated when the canvas needs to be redrawn. It can be NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLCanvas** element handle all attributes defined for a conventional canvas, see IupCanvas.

Apart from these attributes, **IupGLCanvas** handle specific attributes used to define the kind of buffer to be instanced. Such attributes are all **creation only** attributes and must be set before the element is mapped on the native system. After the mapping, specifying these special attributes has no effect.

ACCUM_RED_SIZE, **ACCUM_GREEN_SIZE**, **ACCUM_BLUE_SIZE** and **ACCUM_ALPHA_SIZE**: Indicate the number of bits for representing the color components in the accumulation buffer. Value 0 means the accumulation buffer is not necessary. Default is 0.

ALPHA_SIZE: Indicates the number of bits for representing each colors alpha component (valid only for RGBA and for hardware that store the alpha component). Default is "0".

ARBCONTEXT (non inheritable): enable the usage of ARB extension contexts. If during map the ARB extensions could not be loaded the attribute will be set to NO and the standard context creation will be used. Default: NO. (since 3.6)

BUFFER: Indicates if the buffer will be single "SINGLE" or double "DOUBLE". Default is "SINGLE".

BUFFER_SIZE: Indicates the number of bits for representing the color indices (valid only for INDEX). The system default is 8 (256-color palette).

COLOR: Indicates the color model to be adopted: "INDEX" or "RGBA". Default is "RGBA".

COLORMAP (read-only)**:** Returns "Colormap" in UNIX and "HPALETTE" in Win32, if COLOR=INDEX.

CONTEXT (read-only)**:** Returns "GLXContext" in UNIX and "HGLRC" in Win32.

CONTEXTFLAGS (non inheritable): Context flags. Can be DEBUG, FORWARDCOMPATIBLE or DEBUGFORWARDCOMPATIBLE. Used only when ARBCONTEXT=Yes. (since 3.6)

CONTEXTPROFILE (non inheritable): Context profile mask. Can be CORE, COMPATIBILITY or CORECOMPATIBILITY. Used only when ARBCONTEXT=Yes. (since 3.6)

CONTEXTVERSION (non inheritable): Context version number in the format "major.minor". Used only when ARBCONTEXT=Yes. (since 3.6)

DEPTH_SIZE: Indicates the number of bits for representing the *z* coordinate in the z-buffer. Value 0 means the z-buffer is not necessary.

ERROR (read-only): If an error is found during **IupMap** and **IupGLMakeCurrent**, returns a string containing a description of the error in English. See notes bellow.

LASTERROR (read-only) [Windows Only]: If an error is found, returns a string with the system error description. (Since 3.6)

RED_SIZE, **GREEN_SIZE** and **BLUE_SIZE**: Indicate the number of bits for representing each color component (valid only for RGBA). The system default is usually 8 for each component (True Color support).

REFRESHCONTEXT (write-only) [Windows Only]: action attribute to refresh the internal device context when it is not owned by the window class. The **IupCanvas** of the Win32 driver will always create a window with an owned DC, but GTK in Windows will not. (since 3.0)

STENCIL_SIZE: Indicates the number of bits in the stencil buffer. Value 0 means the stencil buffer is not necessary. Default is 0.

STEREO: Creates a stereo GL canvas (special glasses are required to visualize it correctly). Possible values: "YES" or "NO". Default: "NO". When this flag is set to Yes but the OpenGL driver does not support it, the map will be successful and STEREO will be set to NO and ERROR will not be set (since 3.9).

SHAREDCONTEXT: name of another **IupGLCanvas** that will share its display lists and textures. That canvas must be mapped before this canvas.

VISUAL (read-only)**:** Returns "XVisualInfo*" in UNIX and "HDC" in Win32.

Callbacks

The **IupGLCanvas** element understands all callbacks defined for a conventional canvas, see IupCanvas.

Additionally:

RESIZE_CB: By default the resize callback sets:

```
glViewport(0,0,width,height);
```

SWAPBUFFERS_CB: action generated when **IupGLSwapBuffers** is called. (since 3.11)

```
int function(Ihandle* ih);  [in C]
elem:swapbuffers_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Auxiliary Functions

These are auxiliary functions based on the WGL and XGL extensions. Check the respective documentations for more information. ERROR attribute will be set to "Failed to set new current context." if the call failed. It will reset ERROR to NULL if successful.

```
void IupGLMakeCurrent(Ihandle* ih); [in C]
iup.GLMakeCurrent(ih: ihandle) [in Lua]
or ih:MakeCurrent() [in Lua]
```

Activates the given canvas as the current OpenGL context. All subsequent OpenGL commands are directed to such canvas. The first call will set the global attributes GL_VERSION, GL_VENDOR and GL_RENDERER (since 3.16).

```
int IupGLIsCurrent(Ihandle* ih); [in C]
iup.GLIsCurrent(ih: ihandle) -> status: boolean [in Lua]
or ih:IsCurrent() -> status: boolean [in Lua]
```

Returns a non zero value if the given canvas is the current OpenGL context.

```
void IupGLSwapBuffers(Ihandle* ih); [in C]
iup.GLSwapBuffers(ih: ihandle) [in Lua]
or ih:SwapBuffers() [in Lua]
```

Makes the BACK buffer visible. This function is necessary when a double buffer is used.

```
void IupGLPalette(Ihandle* ih, int index, float r, float g, float b); [in C]
iup.GLPalette(ih: ihandle, index, r, g, b: number) [in Lua]
or ih:Palette(index, r, g, b: number) [in Lua]
```

Defines a color in the color palette. This function is necessary when INDEX color is used.

```
void IupGLUseFont(Ihandle* ih, int first, int count, int list_base); [in C]
iup.GLUseFont(ih:ihandle, first, count, list_base: number) [in Lua]
or ih:UseFont(first, count, list_base: number) [in Lua]
```

Creates a bitmap display list from the current FONT attribute. See the documentation of the wglUseFontBitmaps and glXUseXFont functions. (since 3.0)

```
void IupGLWait(int gl) [in C]
iup.GLWait(gl: number) [in Lua]
```

If gl is non zero it will call glFinish or glXWaitGL, else will call GdiFlush or glXWaitX. (since 3.0)

Notes

In Windows XP, if the COMPOSITE attribute is enabled then the hardware acceleration will be disabled.

The **IupGLCanvas** works with the GTK base driver in Windows and in UNIX (X-Windows).

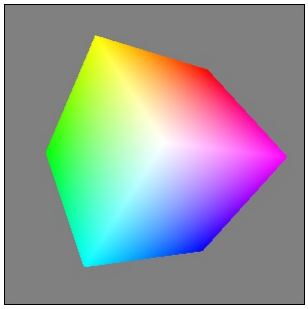
Not available in our SunOS510x86 pre-compiled binaries just because we were not able to compile OpenGL code in our installation.

Possible ERROR strings during **IupMap**:

```
"X server has no OpenGL GLX extension." - OpenGL not supported (UNIX Only)
"No appropriate visual." - Failed to choose a Visual (UNIX Only)
"No appropriate pixel format." - Failed to choose a Pixel Format (Windows Only)
"Could not create a rendering context." - Failed to create the OpenGL context. (Windows and UNIX)
```

Examples

Browse for Example Files



See Also

IupCanvas

IupGLControls (since 3.11)

OpenGL Controls Library

This library contains several controls that behave much like their standard controls counterpart such as **IupLabel**, **IupButton**, **IupFrame**, and so on. But they were designed to be used only embedded in an OpenGL canvas along with the application drawing. They will work and be displayed on top of the application drawing and respond to mouse events concurrently with the application mouse events.

In order to use these controls the application must use the **IupGLCanvasBox** controls instead of the **IupGLCanvas** control. Actually the **IupGLCanvasBox** inherits from the **IupGLCanvas** control so their usage is identical. But **IupGLCanvasBox** can have children and it will manage the display and mouse events of all its children.

All the **IupGlControls** visible elements that can be an embedded children of a **IupGLCanvasBox** are based on the **IupGLSubCanvas** control. Other IUP elements can also be used. It will work seemsly elements that are void containers can also be used, such as **IupHbox**, **IupVbox**, **IupGridBox**, and so on. **IupFill** can also be used, but native elements can also be placed on top although they will not be clipped by **IupGLFrame** and other **IupGlControls** containers. All functions and resources, like **IupImage**, are used just like any other IUP control.

These controls are drawn by IUP using OpenGL on a IupGLCanvas control, and are not native controls.

The **iupglcontrols.h** file must be included in the source code. If you plan to use the control in Lua, you should also include **iupluaglcontrols.h**.

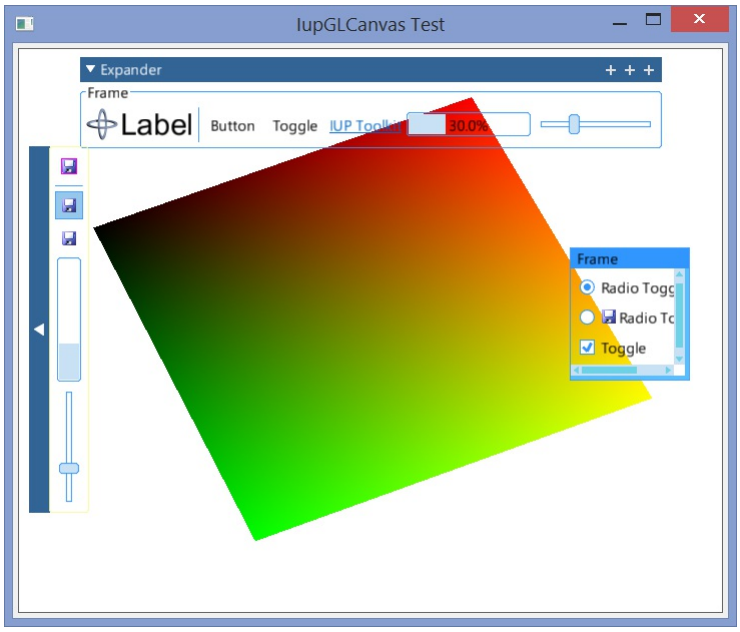
The **IupGLControlsOpen** function must be called after **IupOpen**. To make the controls available in Lua use require"iupluaglcontrols" or manually call the initialization function in C, **iupglcontrolslua_open**, after calling **iuplua_open**.

When manually calling the function your application must be linked to the control library (**iupglcontrols**), the **IupGLCanvas** control library (**iupgl**), with the FTGL library, and with the OpenGL library. To use its bindings to Lua, the program must also be linked to the **iupluaglcontrols** library.

The FTGL library is dependent also on the GLU library. In UNIX, **IupGLControls** is also dependent on **iconv** and **fontconfig**.

Examples

Browse for Example Files



child, ... : List of the identifiers that will be placed in the box. NULL must be used to define the end of the list in C. It can be empty in C or Lua, not in LED.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLCanvasBox** element handle all attributes defined for an OpenGL canvas and a conventional canvas, see IupGLCanvas and IupCanvas.

MARGIN (non inheritable): Defines a margin in pixels. Its value has the format "*width*x*height*", where *width* and *height* are integer values corresponding to the horizontal and vertical margins, respectively. Default: "0x0" (no margin).

REDRAW (non inheritable, write-only): force a full redraw of all elements and the main canvas.

CLIENTSIZE, CLIENTOFFSET: also accepted.

Attributes (at Children)

HORIZONTALALIGN (non inheritable) **(at children only)**: Horizontally aligns the element inside the box. Possible values: "ALEFT", "ACENTER", "ARIGHT" or "FLOAT". Default: "FLOAT". When FLOAT is used its horizontal position is obtained from the POSITION attribute.

VERTICALALIGN (non inheritable) **(at children only)**: Vertically aligns the element inside the box. Possible values: "ATOP", "ACENTER", "ABOTTOM" or "FLOAT". Default: "FLOAT". When FLOAT is used its vertical position is obtained from the POSITION attribute.

EXPANDHORIZONTAL (non inheritable) **(at children only)**: Expand the horizontal size of the element to the box width. Can be Yes or No. Default: No. (since 3.13)

EXPANDVERTICAL (non inheritable) **(at children only)**: Expand the vertical size of the element to the box height. Can be Yes or No. Default: No. (since 3.13)

Callbacks

The **IupGLCanvasBox** element understands all callbacks defined for the IupGLCanvas.

But since it has to forward the mouse and action callbacks to the **IupGLControls** elements when it is mapped the callbacks ACTION, BUTTON_CB, MOTION_CB, WHEEL_CB, and LEAVEWINDOW_CB are replaced by internal callbacks. The application callbacks will still be called and they can be retrieved by using the prefix "APP_" on the callback name. If for some reason the application set one of these callbacks after being mapped, the box internal callback can be retrieved by using the prefix "GLBOX_" on the callback name.

Keyboard focus is NOT processed for **IupGLCanvasBox** children.

The **IupGLCanvas** SWAPBUFFERS_CB callback is used internally to enable a correct display for the box children. This means that when the application calls **IupGLSwapBuffers**, then children will be drawn before actually swapping the double buffer.

Examples

Browse for Example Files

See Also

IupGLCanvas, IupCanvas

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BGCOLOR: background color used in derived controls. Can have an alpha component. Default: NULL.

BORDERCOLOR: color used for borders in derived controls. Can have an alpha component. Default: "50 150 255".

BORDERWIDTH: line width used for borders in derived controls. Default: "1". Any borders can be disabled by simply setting this value to 0.

CURSOR (non inheritable): Defines a cursor for the sub-canvas.

FONT: Uses the FTGL library to render text. Depends on locating a font file that matches the font attribute. See Notes bellow. The default font typeface is changed to Helvetica to avoid system fonts that are not well processed by FreeType.

HIGHLIGHT (non inheritable): flag indicating that the control is highlighted. Dynamically updated during mouse move.

HLCOLOR: color used to indicate a highlight state in derived controls. Can have an alpha component. Default: "200 225 245".

PSCOLOR: color used to indicate a press state in derived controls. Can have an alpha component. Default: "150 200 235".

PRESSED (non inheritable): flag indicating that the control is pressed with button1. Dynamically updated during button press.

REDRAWALL (non inheritable): flag to control the redraw update during a change of state like highlight or pressed. If "No" only the element is redrawn, else all the **IupGLCanvasBox** is redrawn. It will work only if the control is fully opaque. Default: "Yes".

REDRAWFRONT (non inheritable, write-only): redraw only the control on the front buffer. It will work only if the control is fully opaque.

UNDERLINE (non inheritable): flag indicating that the control text should be redrawn with an underline. Since FTGL does not supports underline, the drawing of the text will manually draw a line under the text.

WID (non inheritable): returns the same value as the **IupGLCanvasBox** where the element is inside.

ZORDER (non inheritable, write-only): change the order of the control inside its parent.

ACTIVE, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, TIP, SIZE, RASTERSIZE, VISIBLE: also accepted.

Callbacks

GL_ACTION: Action generated when the sub-canvas needs to be redrawn.

```
int function(Ihandle *ih); [in C]
elem:action() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

GL_BUTTON_CB: Action generated when any mouse button is pressed or released. Same parameters as BUTTON_CB. If IUP_CONTINUE is returned the application callback is called even the user clicked on the sub-canvas.

GL_ENTERWINDOW_CB: Action generated when the mouse enters the element. Same parameters as ENTERWINDOW_CB.

GL_LEAVEWINDOW_CB: Action generated when the mouse leaves the element. Same parameters as LEAVEWINDOW_CB.

GL_MOTION_CB: Action generated when the mouse is moved. Same parameters as MOTION_CB. If IUP_CONTINUE is returned the application callback is called even the user moved the cursor on the sub-canvas.

GL_WHEEL_CB: Action generated when the mouse wheel is rotated. Same parameters as WHEEL_CB. If IUP_CONTINUE is returned the application callback is called even the user clicked on the sub-canvas.

MAP_CB, UNMAP_CB, DESTROY_CB: common callbacks are supported.

Notes

FTGL is used to draw text in OpenGL. This is a third party library not developed at Tecgraf. But its license is also free and have the same freedom as the Tecgraf Library License. You can read the FTGL license and copyright in the file ftgl.txt. FTGL is copyright to Henry Maddocks.

IUP uses the same FTGL library included in the CD library. Currently CD is using the FTGL version 2.1.3-rc5 with modifications.

To locate font files we use several strategies.

1. search for the font in the system. In Windows use the Registry to locate the font, in UNIX use the FontConfig library;
2. use the type face as a file title, compose with the font path to get a filename (assume style already in the typeface);
3. try some pre-defined names, and use the style to compose the filename;
4. use the typeface directly as the file name;

It will search for TrueType (*.ttf) and OpenType (*.otf) font files. It will search in the current directory; in the path returned by the FREETYPEFONTS_DIR environment variable or from the FREETYPEFONTS_DIR global attribute; and in Windows on the Fonts folder.

FTGL fonts are cached internally to optimal use of multiple fonts in the same **IupGLCanvasBox**.

We use OpenGL textures to draw images, so the image width and height MUST be a power of two if OpenGL version is 1.x, modern OpenGL does not have this limitation.

See Also

IupCanvas

IupGLButton (since 3.11)

Creates an embedded OpenGL interface element that is a button. When selected, this element activates a function in the application. Its visual presentation can contain a text and/or an image. It inherits from IupGLLabel. It exists only inside an IupGLCanvasBox.

Creation

```
Ihandle* IupGLButton(const char *title); [in C]
iup.glbutton{[title = title: string]} -> elem: ihandle [in Lua]
glbutton(title) [in LED]
```

title: Text to be shown to the user. It can be NULL. It will set the TITLE attribute.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLButton** element handle all attributes defined for the IupGLLabel control, and consequently for the IupGLSubCanvas control too.

BACKIMAGE (non inheritable): image name to be used as background. It will be zoomed to fill the background (it does not includes the border). Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. (since 3.11.2)

BACKIMAGEHIGHLIGHT (non inheritable): background image name of the element in highlight state. If it is not defined then the BACKIMAGE is used. (since 3.11.2)

BACKIMAGEINACTIVE (non inheritable): background image name of the element when inactive. If it is not defined then the BACKIMAGE is used and its colors will be replaced by a modified version creating the disabled effect. (since 3.11.2)

BACKIMAGEPRESS (non inheritable): background image name of the element in pressed state. If it is not defined then the BACKIMAGE is used.(since 3.11.2)

FGCOLOR: Text color. Can have an alpha component. Default: "0 0 0". If TITLE and IMAGE are both not defined then the button will show a color in a rectangle using this attribute (since 3.15).

FRONTIMAGE (non inheritable): image name to be used as foreground. It will be zoomed to fill the foreground (it does not includes the border). The foreground has the same are as the background, but it is drawn at last. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. (since 3.11.2)

FITTOBACKIMAGE (non inheritable): enable the natural size to be computed from the BACKIMAGE. If BACKIMAGE is not defined will be ignored. When set to Yes it will set BORDERWIDTH to 0. Can be Yes or No. Default: No. (since 3.11.2)

FRONTIMAGEHIGHLIGHT (non inheritable): foreground image name of the element in highlight state. If it is not defined then the FRONTIMAGE is used. (since 3.11.2)

FRONTIMAGEINACTIVE (non inheritable): foreground image name of the element when inactive. If it is not defined then the FRONTIMAGE is used and its colors will be replaced by a modified version creating the disabled effect. (since 3.11.2)

FRONTIMAGEPRESS (non inheritable): foreground image name of the element in pressed state. If it is not defined then the FRONTIMAGE is used.(since 3.11.2)

Callbacks

The **IupGLButton** element handle all callbacks defined for the **IupGLSubCanvas** control.

ACTION: Action generated when the button 1 (usually left) is selected. This callback is called only after the mouse is released and when it is released inside the button area.

```
int function(Ihandle* ih); [in C]
elem:action() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Returns: IUP_CLOSE will be processed.

Notes

The difference between a **IupGLButton** and a **IupGLLabel** is the border (controlled by **IupGLSubCanvas** attributes BORDERWIDTH and BORDERCOLOR), the change in background color for state feedback (controlled by PRESSCOLOR and HLCOLOR attributes), and the callback to notify the application.

The natural size if the same as a **IupGLLabel** plus BORDERWIDTH.

See Also

IupImage, IupGLToggle, IupGLLabel

IupGLExpander (since 3.11)

Creates an embedded OpenGL container that can interactively show or hide its child. It inherits from IupGLSubCanvas. It exists only inside an IupGLCanvasBox.

Creation

```
Ihandle* IupGLExpander(Ihandle* child); [in C]
iup.glexpander{child: ihandle} -> (elem: ihandle) [in Lua]
glexpander(child) [in LED]
```

child: Identifier of an interface element. It can be NULL (nil in Lua), not optional in LED.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLExpander** element handle all attributes defined for a IupGLSubCanvas control.

BACKCOLOR (non inheritable): background color of the title area. Default: "50 100 150".

BARPOSITION: indicates the bar handler position. Possible values are "TOP", "BOTTOM", "LEFT" or "RIGHT". Default: "TOP".

BARSIZE (non inheritable): controls the size of the bar handler. Default: the height of a text line plus 5 pixels.

EXPAND (non inheritable): the default value is "YES".

EXTRABUTTONS (non inheritable): sets the number of extra image buttons at right when BARPOSITION=TOP. The maximum number of buttons is 3. See the EXTRABUTTON_CB callback. Default: 0.

IMAGEEXTRAid: image name used for the button. id can be 1, 2 or 3. 1 is the rightmost button, and count from right to left.
IMAGEEXTRAPRESSid: image name used for the button when pressed.
IMAGEEXTRAHIGHLIGHTid: image name for the button used when mouse is over the button area.

All images must be 16x16, or smaller but the occupied size will still be 16x16.

FORECOLOR (non inheritable): text and arrow color. Default: "255 255 255".

HIGHCOLOR (non inheritable): text and arrow color when highlight. Default: "200 225 245".

IMAGE (non inheritable): image name to replace the arrow by an image STATE=CLOSE. Works only when BARPOSITION=TOP. The minimum horizontal space reserved for the handler is 20 pixels. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage.

IMAGEOPEN: image name used when STATE=OPEN.
IMAGEHIGHLIGHT: image name used when mouse is over the bar handler and STATE=CLOSE.
IMAGEOPENHIGHLIGHT: image name used when mouse is over the bar handler and STATE=OPEN.

MOVEABLE (non inheritable): enable the frame to be interactively moved when it is a direct child of the **IupGLCanvasBox**. Default: NO.

MOVETOTOP (non inheritable): when MOVEABLE=Yes and the frame is moved then its ZORDER is also set to TOP. (Since 3.11.1)

PRESSCOLOR (non inheritable): text and arrow color when pressed. Default: "150 200 235".

STATE (non inheritable): Show or hide the container elements. Possible values: "OPEN" (expanded) or "CLOSE" (collapsed). Default: OPEN. Setting this attribute will automatically change the layout of the entire dialog so the child can be recomposed.

TITLE (non inheritable): title text, shown in the title bar near the expand or collapse symbol. Shown only when BARPOSITION=TOP.

TITLEBACKIMAGE (non inheritable): image name to be used as a background of the title area. It will be zoomed to fill the background (it does not includes the border). Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. (Since 3.11.1)

TITLEBACKIMAGEINACTIVE (non inheritable): background image name of the element when inactive. If it is not defined then the TITLEBACKIMAGE is used and its colors will be replaced by a modified version creating the disabled effect. (Since 3.11.2)

CLIENTSIZE, CLIENTOFFSET: also accepted.

Callbacks

The **IupGLFrame** element handle all callbacks defined for the **IupGLSubCanvas** control.

ACTION: Action generated after the expander state is interactively changed.

```
int function(Ihandle* ih); [in C]
elem:action() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

MOVE_CB: Called after the frame was moved on the **IupGLCanvasBox**, when MOVEABLE=Yes. The coordinates are the same as the POSITION attribute.

```
int function(Ihandle *ih, int x, int y); [in C]
elem:trayclick_cb(x, y: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
x, **y**: coordinates of the new position.

OPENCLOSE_CB: Action generated before the expander state is interactively changed.

```
int function(Ihandle* ih, int state); [in C]
elem:openclose_cb(state: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
state: new state to be applied.

Returns: if return IUP_IGNORE the new state is ignored.

EXTRABUTTON_CB: Action generated when any mouse button is pressed and released.

```
int function(Ihandle* ih, int button, int pressed); [in C]
elem:extrabutton_cb(button, pressed: number) -> (ret: number) [in Lua]
```

ih: identifies the element that activated the event.
button: identifies the extra button. can be 1, 2 or 3. (this is not the same as BUTTON_CB)
pressed: indicates the state of the button:

0 - mouse button was released;
1 - mouse button was pressed.

Notes

The container can be created with no elements and be dynamic filled using IupAppend or IupInsert.

When the TITLE is defined and BARPOSITION=TOP then the expand/collapse symbol is left aligned. In all other situations the expand/collapse symbol is centered.

IupGLFrame (since 3.11)

Creates an embedded OpenGL container, which draws a frame with a title around its child. It inherits from IupGLSubCanvas. It exists only inside an IupGLCanvasBox.

Creation

```
Ihandle* IupGLFrame(Ihandle *child); [in C]
iup.glframe{child: ihandle} -> (elem: ihandle) [in Lua]
glframe(child) [in LED]
```

child: Identifier of an interface element which will receive the frame around. It can be NULL (nil in Lua), not optional in LED.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLFrame** element handle all attributes defined for a IupGLSubCanvas control.

ALIGNMENT (non inheritable): horizontal and vertical alignment. Possible values: "ALEFT", "ACENTER" and "ARIGHT", combined to "ATOP", "ACENTER" and "ABOTTOM". Default: "ACENTER:ACENTER". Partial values are also accepted, like "ARIGHT" or ":ATOP", the other value will be used from the current alignment.

BACKCOLOR (non inheritable): color used as background when TITLE and IMAGE are not defined. Can have an alpha component. Default: NULL. Used instead of BGCOLOR to avoid inheritance problems.

BACKIMAGE (non inheritable): image name to be used as background when TITLE and IMAGE are not defined. It will be zoomed to fill the background (it does not includes the border). Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. (Since 3.11.1)

BACKIMAGEINACTIVE (non inheritable): background image name of the element when inactive. If it is not defined then the BACKIMAGE is used and its colors will be replaced by a modified version creating the disabled effect. (since 3.11.2)

EXPAND (non inheritable): The default value is "YES".

FORECOLOR (non inheritable): Text color. Can have an alpha component. Default: "0 0 0". Used instead of FGCOLOR to avoid inheritance problems.

FRAMECOLOR (non inheritable): color used to draw the frame border. Can have an alpha component. Default: "50 150 255". Used instead of BORDERCOLOR to avoid inheritance problems.

FRAMEWIDTH (non inheritable): line width of the frame border. Default: 1. Used instead of BORDERWIDTH to avoid inheritance problems.

IMAGE (non inheritable): Image name. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage.

IMAGEHIGHLIGHT (non inheritable): Image name of the element in highlight state. If it is not defined then the IMAGE is used.

IMAGEINACTIVE (non inheritable): Image name of the element when inactive. If it is not defined then the IMAGE is used and its colors will be replaced by a modified version creating the disabled effect.

IMAGEPRESS (non inheritable): Image name of the element in pressed state. If it is not defined then the IMAGE is used.

IMAGEPOSITION (non inheritable): Position of the image relative to the text when both are defined. Can be: LEFT, RIGHT, TOP, BOTTOM. Default: LEFT.

MOVEABLE (non inheritable): enable the frame to be interactively moved when it is a direct child of the IupGLCanvasBox. Default: NO.

MOVETOTOP (non inheritable): when MOVEABLE=Yes and the frame is moved then its ZORDER is also set to TOP. (Since 3.11.1)

PADDING (non inheritable): internal margin for the title area. Default value: "2x0".

SPACING (non inheritable): defines the spacing between the image and the title. Default: "2".

TITLE (non inheritable): Text the user will see at the top of the frame.

TITLEBACKIMAGE (non inheritable): image name to be used as background on the title area. Works only when TITLEBOX=Yes. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. (Since 3.11.1)

TITLEBACKIMAGEINACTIVE (non inheritable): background image name of the element when inactive. If it is not defined then the TITLEBACKIMAGE is used and its colors will be replaced by a modified version creating the disabled effect. (Since 3.11.2)

TITLEBOX (non inheritable): enable a different visual style for the frame. Instead of the traditional round frame that starts and ends at the title area, it will draw a filled box for the title area and a regular rectangle around the child. Default: NO.

TITLEOFFSET (non inheritable): horizontal offset from the left border to start the title area. Default: 5.

CLIENTSIZE, CLIENTOFFSET: also accepted.

Callbacks

The **IupGLFrame** element handle all callbacks defined for the **IupGLSubCanvas** control.

MOVE_CB: Called after the frame was moved on the **IupGLCanvasBox**, when MOVEABLE=Yes. The coordinates are the same as the POSITION attribute.

```
int function(Ihandle *ih, int x, int y); [in C]
elem:trayclick_cb(x, y: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
x, **y**: coordinates of the new position.

Notes

The **IupGLFrame** can contain text and image simultaneously at the title area.

The frame can be created with no elements and be dynamic filled using IupAppend or IupInsert.

IupGLLabel (since 3.11)

Creates an embedded OpenGL label interface element, which displays a text and/or an image. It inherits from IupGLSubCanvas. It exists only inside an IupGLCanvasBox.

Creation

```
Ihandle* IupGLLabel(const char *title); [in C]
iup.gllabel{[title = title: string]} -> (elem: ihandle) [in Lua]
gllabel(title) [in LED]
```

title: Text to be shown on the label. It can be NULL. It will set the TITLE attribute.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLLabel** element handle all attributes defined for a IupGLSubCanvas control.

ACTIVE: Since it has no callbacks, the only difference between an active label and an inactive one is its visual feedback. Possible values: "YES, "NO". Default: "YES".

ALIGNMENT (non inheritable): horizontal and vertical alignment. Possible values: "ALEFT", "ACENTER" and "ARIGHT", combined to "ATOP", "ACENTER" and "ABOTTOM". Default: "ACENTER:ACENTER". Partial values are also accepted, like "ARIGHT" or ":ATOP", the other value will be used from the current alignment.

FGCOLOR: Text color. Can have an alpha component. Default: "0 0 0".

IMAGE (non inheritable): Image name. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage.

IMAGEHIGHLIGHT (non inheritable): Image name of the element in highlight state. If it is not defined then the IMAGE is used.

IMAGEINACTIVE (non inheritable): Image name of the element when inactive. If it is not defined then the IMAGE is used and its colors will be replaced by a modified version creating the disabled effect.

IMAGEPRESS (non inheritable): Image name of the element in pressed state. If it is not defined then the IMAGE is used.

IMAGEPOSITION (non inheritable): Position of the image relative to the text when both are defined. Can be: LEFT, RIGHT, TOP, BOTTOM. Default: LEFT.

PADDING (non inheritable): internal margin. Works just like the MARGIN attribute of the **IupHbox** and **IupVbox** containers, but uses a different name to avoid inheritance problems. Default value: "0x0".

SPACING (non inheritable): defines the spacing between the image and the title. Default: "2".

TITLE (non inheritable): Label's text. The '\n' character is accepted for line change.

Notes

The **IupGLLabel** can contain text and image simultaneously.

The natural size will be a combination of the size of the image and the title, if any, plus PADDING and SPACING (if both image and title are present).

See Also

IupImage, IupGLButton, IupGLToggle.

IupGLLink (since 3.11)

Creates an embedded OpenGL label that displays an underlined clickable text. It inherits from IupGLLabel. It exists only inside an IupGLCanvasBox.

Creation

```
Ihandle* IupGLLink(const char *url, const char * title); [in C]
iup.gllink{[url = url: string], [title = title: string]} -> (elem: ihandle) [in Lua]
gllink(url, title) [in LED]
```

url: the destination address of the link. Can be any text. If **IupHelp** is used should be a valid URL. It can be NULL. It will set the URL attribute.
title: Text to be shown on the link. It can be NULL. It will set the TITLE attribute.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLLink** element handle all attributes defined for the IupGLLabel control, and consequently for the IupGLSubCanvas control too.

FGCOLOR: Text color. Default: the global attribute LINKFGCOLOR.

URL: the default value is "YES".

Callbacks

The **IupGLLink** element handle all callbacks defined for the **IupGLSubCanvas** control.

ACTION: Action generated when the link is activated.

```
int function(Ihandle* ih, char *url); [in C]
elem:action(url: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
url: the destination address of the link.

Returns: IUP_CLOSE will be processed. If returns IUP_DEFAULT or it is not defined, the **IupHelp** function will be called.

Notes

When the cursor is over the text, it is changed to the HAND cursor.

If the callback is not defined the **IupHelp** function is called with the given URL.

See Also

IupGLLabel, IupHelp.

IupGLProgressBar (since 3.11)

Creates an embedded OpenGL progress bar control. Shows a percent value that can be updated to simulate a progression. It inherits from IupGLSubCanvas. It exists only inside an IupGLCanvasBox.

Creation

```
Ihandle* IupGLProgressBar(void); [in C]
iup.glprogressbar{} -> (elem: ihandle) [in Lua]
glprogressbar() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLProgressBar** element handle all attributes defined for a IupGLSubCanvas control.

BACKIMAGE (non inheritable): image name to be used as background. It will be zoomed to fill the background (it does not includes the border). Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. (since 3.11.2)

BACKIMAGEHIGHLIGHT (non inheritable): background image name of the element in highlight state. If it is not defined then the BACKIMAGE is used. (since 3.11.2)

BACKIMAGEINACTIVE (non inheritable): background image name of the element when inactive. If it is not defined then the BACKIMAGE is used and its colors will be replaced by a modified version creating the disabled effect. (since 3.11.2)

BACKIMAGEPRESS (non inheritable): background image name of the element in pressed state. If it is not defined then the BACKIMAGE is used.(since 3.11.2)

FGCOLOR: Controls the bar color. Can have an alpha component. Default: "200 225 245".

FITTOBACKIMAGE (non inheritable): enable the natural size to be computed from the BACKIMAGE. If BACKIMAGE is not defined will be ignored. When set to Yes it will set BORDERWIDTH to 0. Can be Yes or No. Default: No. (since 3.11.2)

MAX (non inheritable): Contains the maximum value. Default is "1".

MIN (non inheritable): Contains the minimum value. Default is "0".

ORIENTATION: can be "VERTICAL" or "HORIZONTAL". Default: "HORIZONTAL". Horizontal goes from left to right, and vertical from bottom to top.

PADDING: internal margin. Works just like the MARGIN attribute of the **IupHbox** and **IupVbox** containers, but uses a different name to avoid inheritance problems. Default value: "0x0".

SHOWTEXT: Indicates if the text inside the bar is to be shown or not. Possible values: "YES" or "NO". Default: "YES".

TEXT (non inheritable): Contains a text to be shown inside the bar when SHOW_TEXT=YES. If it is NULL, the percentage calculated from VALUE will be used.

TXTCOLOR: Text color. Can have an alpha component. Default: "0 0 0".

VALUE (non inheritable): Contains a number between "MIN" and "MAX", controlling the current position.

Notes

The natural size is the height of one character in one direction and the width of 15 characters in the other, plus PADDING and BORDERWIDTH.

IupGLScrollBox (since 3.11)

Creates an embedded OpenGL container that allows its child to be scrolled. It inherits from IupGLSubCanvas. It exists only inside an IupGLCanvasBox.

Creation

```
Ihandle* IupGLScrollBox(Ihandle* child); [in C]
iup.glscrollbox{child: ihandle} -> (elem: ihandle) [in Lua]
glscrollbox(child) [in LED]
```

child: Identifier of an interface element which will receive the box. It can be NULL (nil in Lua), not optional in LED.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLScrollBox** element handle all attributes defined for a IupGLSubCanvas control.

It contains automatic scrollbars that are shown or hidden accordingly if the child natural size fits the client size. See the Scrollbars Attributes for **IupGLControls** for more details.

EXPAND (non inheritable): The default value is "YES".

POSX: Position of the thumb in the horizontal scrollbar. Default: "0.0". DX is the visible horizontal area and XMAX is set to the child natural width.

POSY: Position of the thumb in the vertical scrollbar. Default: "0.0". DY is the visible vertical area and YMAX is set to the child natural height.

CLIENTSIZE, CLIENTOFFSET: also accepted.

Notes

The box allows the application to create a virtual space for the dialog that is actually larger than the visible area. The current size of the box defines the visible area. The natural size of the child (and its children) defines the virtual space size. So the **IupGLScrollBox** does not depend on its child size or expansion, and its natural size is always 0x0.

The user can move the box contents by dragging the background. Also the mouse wheel scrolls the contents vertically.

The box can be created with no elements and be dynamic filled using IupAppend or IupInsert.

Examples

Browse for Example Files

Scrollbars Attributes for IupGLControls (since 3.11)

Scrollbars are always enabled and they are always automatically shown or hidden accordingly to D* >= *MAX-*MIN.

Configuration Attributes (non inheritable)

DX: Size of the thumb in the horizontal scrollbar. Also the horizontal page size. Default: "10".

DY: Size of the thumb in the vertical scrollbar. Also the vertical page size. Default: "10".

POSX: Position of the thumb in the horizontal scrollbar. Default: "0".

POSY: Position of the thumb in the vertical scrollbar. Default: "0".

XMIN: Minimum value of the horizontal scrollbar. Default: "0".

XMAX: Maximum value of the horizontal scrollbar. Default: "100".

YMIN: Minimum value of the vertical scrollbar. Default: "0".

YMAX: Maximum value of the vertical scrollbar. Default: "100".

LINEX: The amount the thumb moves when an horizontal step is performed. Default: 1/10th of DX.

LINEY: The amount the thumb moves when a vertical step is performed. Default: 1/10th of DY.

Appearance Attributes (non inheritable)

BACKCOLOR (non inheritable): color used as background. Can have an alpha component. Default: "200 225 245". Used instead of BGCOLOR to avoid inheritance problems.

FORECOLOR (non inheritable): handler and arrow color. Can have an alpha component. Default: "110 210 230". Used instead of FGCOLOR to avoid inheritance problems.

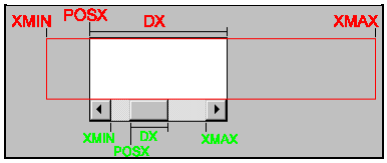
HIGHCOLOR (non inheritable): handler and arrow color when highlight. Default: "90 190 255".

PRESSCOLOR (non inheritable): handler and arrow color when pressed. Default: "50 150 255".

SCROLLBARSIZE: The width of the vertical scrollbar or the height of the horizontal scrollbar. Default: 11.

Notes

The scrollbar allows you to create a virtual space associated to the element. In the image below, such space is marked in **red**, as well as the attributes that affect the composition of this space. In **green** you can see how these attributes are reflected on the scrollbar.



Hence you can clearly deduce that POSX is limited to XMIN and XMAX-DX, or **XMIN<=POSX<=XMAX-DX**.

IMPORTANT: set XMAX to the integer size of the virtual space, NOT to "width-1", or the last pixel of the virtual space will never be visible. If you decide to let XMAX with the default value of 100 and to control only DX, then use the formula DX=visible_width/width.

When the virtual space has the same size as the canvas, DX equals XMAX-XMIN, the scrollbar is automatically hidden. The width of the vertical scrollbar (the same as the height of the horizontal scrollbar) can be obtained using the SCROLLBARSIZE attribute.

The same is valid for YMIN, YMAX, DY and POSY. But remember that the Y axis is oriented from top to bottom in IUP. So if you want to consider YMIN and YMAX as bottom-up oriented, then the actual YPOS must be obtained using **YMAX-DY-POSY**.

If you have to change the properties of the scrollbar (XMIN, XMAX and DX) but you want to keep the thumb still (if possible) in the same relative position, then you have to also recalculate its position (POSX) using the old position as reference to the new one. For example, you can convert it to a 0-1 interval and then scale to the new limits:

```
old_posx_relative = (old_posx - old_xmin)/(old_xmax - old_xmin)
posx = (xmax - xmin)*old_posx_relative + xmin
```

Affects

IupGLScrollBox

See Also

POSX, XMIN, XMAX, DX, POSY, YMIN, YMAX, DY

IupGLSeparator (since 3.11)

Creates an embedded OpenGL separator interface element, which displays a vertical or horizontal line. It inherits from IupGLSubCanvas. It exists only inside an IupGLCanvasBox.

Creation

```
Ihandle* IupGLSeparator(void); [in C]
iup.glseparator{} -> (elem: ihandle) [in Lua]
glseparator() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLSeparator** element handle all attributes defined for a IupGLSubCanvas control.

ORIENTATION: can be "VERTICAL" or "HORIZONTAL". Default: "VERTICAL".

Notes

The **IupGLSeparator** visual is controlled by **IupGLSubCanvas** attributes BORDERWIDTH and BORDERCOLOR.

The natural size will be BORDERWIDTH in one direction and it will expand if there is free space in the other direction.

IupGLSizebox (since 3.11)

Creates a void container that allows its child to be resized. Allows expanding and contracting the child **size** in one or two directions. It inherits from IupGLSubCanvas. It exists only inside an IupGLCanvasBox.

Creation

```
Ihandle* IupGLSizebox(Ihandle* child); [in C]
iup.glsizebox{child: ihandle} -> (elem: ihandle) [in Lua]
glsizebox(child) [in LED]
```

child: Identifier of an interface element which will receive the box. It can be NULL (nil in Lua), not optional in LED.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLScrollBox** element handle all attributes defined for a IupGLSubCanvas control.

FORECOLOR: Changes the color of the bar handler. The value should be given in "R G B" color style. Default: "192 192 192".

RESIZERS: Indicates the direction of the resize. Possible values are "VERTICAL", "HORIZONTAL", or "BOTH". Default: "BOTH". The handler is always placed at the right/bottom of its child.

EXPAND (non inheritable): The default value is "YES".

CLIENTSIZE, CLIENTOFFSET: also accepted.

Notes

The control inside the **IupGLSizeBox** will have its **User** size changed. See the Layout Guide for mode details on sizes.

IupGLSizeBox can make the layout to be resized larger than the **IupGLCanvasBox** size so some controls will be positioned outside the box area at right or bottom. In fact this is part of the dynamic layout default reposition of controls inside the box. See the **IupRefresh** function. The IUP layout does not have a maximum limit only a minimum, except if you use the MAXSIZE common attribute.

The box can be created with no elements and be dynamic filled using IupAppend or IupInsert.

Examples

Browse for Example Files

IupGLToggle (since 3.11)

Creates an embedded OpenGL toggle interface element. It is a two-state (on/off) button that, when selected, generates an action that activates a function in the associated application. Its visual representation can contain a text and/or an image. It inherits from IupGLButton. It exists only inside an IupGLCanvasBox.

Creation

```
Ihandle* IupGLToggle(const char *title); [in C]
iup.gltoggle{[title = title: string]} -> (elem: ihandle) [in Lua]
gltoggle(title) [in LED]
```

title: Text to be shown on the toggle. It can be NULL. It will set the TITLE attribute.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLToggle** element handle all attributes defined for the IupGLButton control, and consequently for the IupGLLabel control and for the IupGLSubCanvas control too.

CHECKMARK (non inheritable): Enables the check mark. Default: NO. When enabled the border and the background are not drawn, and a check mark box is drawn at left or right, according to RIGHTBUTTON.

CHECKMARKWIDTH (non inheritable): Size of the check mark. Default: 14.

RADIO (read-only): returns if the toggle is inside a radio. Can be "YES" or "NO". Valid only after the element is mapped, before returns NULL.

RIGHTBUTTON (non inheritable): place the check button at the right of the text. Can be "YES" or "NO". Default: "NO".

VALUE (non inheritable): Toggle's state. Values can be "ON", "OFF" or "TOGGLE". Default: "OFF". The TOGGLE option will invert the current state.

Callbacks

The **IupGLToggle** element handle all callbacks defined for the **IupGLSubCanvas** control.

ACTION: Action generated when the toggle's state (on/off) was changed. The callback also receives the toggle's state.

```
int function(Ihandle* ih, int state); [in C]
elem:action(state: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
state: 1 if the toggle's state was set to on; 0 if it was set to off.

Returns: IUP_CLOSE will be processed.

VALUECHANGED_CB: Called after the value was interactively changed by the user. Called after the ACTION callback, but under the same context.

```
int function(Ihandle *ih); [in C]
elem:valuechanged_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Notes

IupGLToggle visual is the same as a **IupGLButton**. There is no check mark.

To build a set of mutual exclusive toggles, insert them in a **IupRadio** container. They must be inserted before creation, and their behavior can not be changed.

A toggle that is a child of an **IupRadio** automatically receives a name when its is mapped into the native system. (since 3.16)

See Also

IupImage, IupGLButton, IupGLLabel, IupRadio.

orientation: optional orientation of valuator. Can be NULL. See ORIENTATION attribute.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupGLVal** element handle all attributes defined for a IupGLSubCanvas control.

BACKIMAGE (non inheritable): image name to be used as background. It will be zoomed to fill the background (it does not includes the border). Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. IMPORTANT: for the slider match the handler movement area the backimage must has a transparent space that will have room for the handler, in the extremes this space should be at least half the handler size. (since 3.11.2)

BACKIMAGEHIGHLIGHT (non inheritable): background image name of the element in highlight state. If it is not defined then the BACKIMAGE is used. (since 3.11.2)

BACKIMAGEINACTIVE (non inheritable): background image name of the element when inactive. If it is not defined then the BACKIMAGE is used and its colors will be replaced by a modified version creating the disabled effect. (since 3.11.2)

BACKIMAGEPRESS (non inheritable): background image name of the element in pressed state. If it is not defined then the BACKIMAGE is used. (since 3.11.2)

FGCOLOR: Controls the bar color. Can have an alpha component. Default: "200 225 245".

FITTOBACKIMAGE (non inheritable): enable the natural size to be computed from the BACKIMAGE. If BACKIMAGE is not defined will be ignored. When set to Yes it will set BORDERWIDTH to 0. Can be Yes or No. Default: No. (since 3.11.2)

HANDLERSIZE (non inheritable): handler size in the same direction of the ORIENTATION. Default: 0. If set to 0 it will be calculated with half of the dimension opposite to the ORIENTATION. If IMAGE is used, it will be ignored (since 3.11.2). When IMAGE is not used the handler size in the opposite direction is the size of the element.

HLCOLOR: color used to indicate a highlight state. Can have an alpha component. Default: "190 210 230".

IMAGE (non inheritable): Image name for the handler. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. If defined the handler will be replaced by the image.

IMAGEHIGHLIGHT (non inheritable): Image name of the element in highlight state. If it is not defined then the IMAGE is used.

IMAGEINACTIVE (non inheritable): Image name of the element when inactive. If it is not defined then the IMAGE is used and its colors will be replaced by a modified version creating the disabled effect.

IMAGEPRESS (non inheritable): Image name of the element in pressed state. If it is not defined then the IMAGE is used.

MAX: Contains the maximum valuator value. Default is "1". When changed the display will not be updated until VALUE is set.

MIN: Contains the minimum valuator value. Default is "0". When changed the display will not be updated until VALUE is set.

ORIENTATION (non inheritable): Informs whether the valuator is "VERTICAL" or "HORIZONTAL". Vertical valuators are bottom to up, and horizontal valuators are left to right variations of min to max. Default: "HORIZONTAL".

SLIDERSIZE (non inheritable): slider size in the same direction of the ORIENTATION. Default: 5. Ignored when BACKIMAGE is used.

VALUE (non inheritable): Contains a number between MIN and MAX, indicating the valuator position. Default: "0.0".

Callbacks

The **IupGLVal** element handle all callbacks defined for the **IupGLSubCanvas** control.

VALUECHANGED_CB: Called after the value was interactively changed by the user.

```
int function(Ihandle *ih); [in C]
elem:valuechanged_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

VALUECHANGING_CB: Called when the value starts or ends to be interactively changed by the user.

```
int function(Ihandle *ih, int start); [in C]
elem:valuechanging_cb(start: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
start: flag that indicates if the value started to be changed (1) or the change just ended (0).

Notes

The natural size is the height of one character in one direction and the width of 15 characters in the other.

See Also

IupCanvas, IupPPlot, IupMglPlot

See Also

IupCanvas, IupMglPlot

Differences from IupPPlot

Uses OpenGL for screen output and internal drivers for metafile output. **IupPPlot** uses CD for screen and metafile output.

Selection and editing of a dataset using the DS_EDIT attribute are not implemented.

All functions use double floating point. (since 3.11)

New support for 3D data and 3D plots. New support for planar and volumetric data. New ALPHA, ANTIALIAS, DS_DIMENSION, LEGENDBOX, BOX, BOXTICKS, BOXCOLOR, AXS_*ORIGIN, AXS_?LABELPOSITION, AXS_?LABELROTATION, AXS_?TICKVALUESROTATION, LEGENDCOLOR, TITLECOLOR, LIGHT, COLORBAR*, COLORSCHEME attributes. Many new DS_MODE options.

USE_IMAGERGB and USE_GDI+ attributes are NOT supported. MARGIN* attributes are NOT supported. AXS_?SCALE attribute does NOT support the LOG2 and LOGN values. The Crosshair cursor is not supported.

The PREDRAW_CB, POSTDRAW_CB callbacks does not includes the CD canvas parameter. *FONTSIZE attributes are a multiple factor of the FONT size. DASH_DOT_DOT line style is not supported, but has new line styles: LONGDASHED, SMALLDASHED and SMALLDASH_DOT. AXS_?TICKMAJORSIZE, MARKSIZE are in normalized coordinates. New options for GRID: Z, XYZ, XZ, YZ. AXS_?TICKSIZE renamed to AXS_?TICKMINORSIZE, and is a factor of the AXS_?TICKMAJORSIZE. AXS_?TICKMAJORSPAN default value is -5. AXS_?TICKFORMAT default is internally

computed according to the Min-Max range.

Function Mapping:

```
IupPPlotBegin  -> IupMglPlotBegin    (IMPORTANT: parameter is the dimension 1, 2 or 3)
IupPPlotAddStr -> IupMglPlotAdd1D
IupPPlotAdd    -> IupMglPlotAdd2D
   (none)         IupMglPlotAdd3D
IupPPlotEnd    -> IupMglPlotEnd
   (none)            IupMglPlotNewDataSet
IupPPlotInsertStr    (not mapped, use IupMglPlotInsert1D)
IupPPlotInsert       (not mapped, use IupMglPlotInsert2D)
IupPPlotInsertStrPoints -> IupMglPlotInsert1D   (names array is optional)
IupPPlotInsertPoints    -> IupMglPlotInsert2D
   (none)                  IupMglPlotInsert3D
IupPPlotAddStrPoints    -> IupMglPlotInsert1D   (insert at DS_COUNT)
IupPPlotAddPoints       -> IupMglPlotInsert2D   (insert at DS_COUNT)
   (none)                  IupMglPlotInsert3D
   (none)                  IupMglPlotSet1D
   (none)                  IupMglPlotSet2D
   (none)                  IupMglPlotSet3D
   (none)                  IupMglPlotSetData
   (none)                  IupMglPlotLoadData
   (none)                  IupMglPlotSetFromFormula
IupPPlotTransform       -> IupMglPlotTransform  (includes z coordinate)
   (none)                  IupMglPlotTransformXYZ
   (cdCanvasMark)       -> IupMglPlotDrawMark
   (cdCanvasLine)       -> IupMglPlotDrawLine
   (cdCanvasText)       -> IupMglPlotDrawText
IupPPlotPaintTo         -> IupMglPlotPaintTo    (parameters are different)
```

Known Issues/To Do

- Add UTF-8 mode using MathGL unicode support.
- Compile MathGL using OpenMP support.
- **Text render quality is lower than in IupPPlot.**
- Logarithm scale is not working properly.
- Automatic ticks computation needs to be improved.
- Text rotation when DS_SHOWVALUES=Yes is not ok. (MathGL)
- When OPENGL=Yes initial size is smaller. (MathGL)
- **There is still lots of MathGL features not available in IupMglPlot.**

See Also

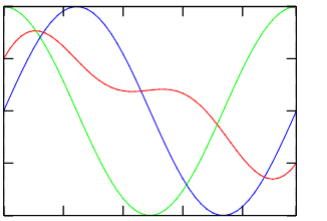
IupCanvas, IupPPlot

IupMglPlot DS_MODES Options

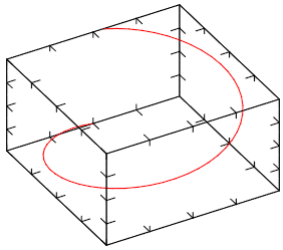
For Linear Datasets

LINE

Draws lines between points. DS_COLOR, DS_LINESTYLE and DS_LINEWIDTH are used to configure the lines.



(3 plots)
BOX=Yes



ROTATE=40:0:60
BOX=Yes

MARK

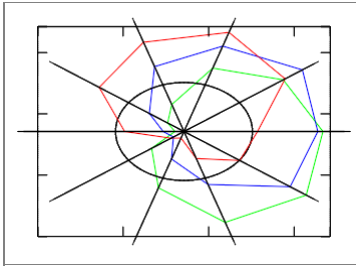
Draws a mark in each point. DS_COLOR, DS_MARKSTYLE and DS_MARKSIZE are used to configure the marks.

MARKLINE

Draws lines between points and draws a mark in each point. Same as if LINE and MARK where set together.

RADAR

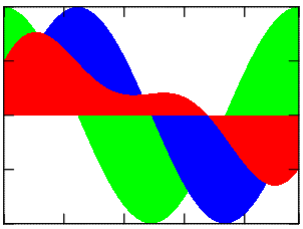
Draws a radar chart. Like a LINE plot in polar coordinates. RADARSHIFT configures an additional radial shift of the data [If rs<0 then rs=max(0, -min(a))], default=-1. If DATAGRID=Yes then a grid of radial lines and a circle for rs are drawn. DS_COLOR, DS_LINESTYLE and DS_LINEWIDTH are used to configure the lines. DS_COLOR, DS_MARKSTYLE and DS_MARKSIZE are used to configure the marks.



(3 plots)
RADARSHIFT=0.4
DATAGRID=Yes
BOX=Yes

AREA

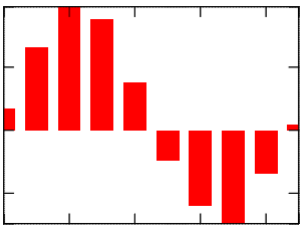
Draws lines between points and fills it to axis plane. DS_COLOR is used to configure fill color. The order of the datasets will define which one will be drawn first.



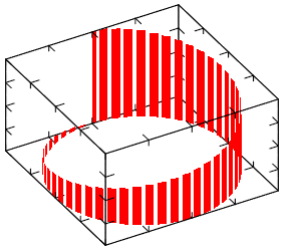
(3 plots)
AXS_XORIGIN=0
AXS_YORIGIN=0
BOX=Yes

BAR

Draws vertical bars from points to axis plane. If DATAGRID=Yes then grid lines are drawn, default=No. BARWIDTH sets relative width of rectangles, default=0.7.



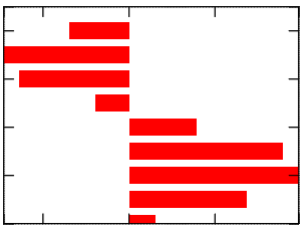
AXS_XORIGIN=0
AXS_YORIGIN=0
BOX=Yes



ROTATE=40:0:60
BOX=Yes

BARHORIZONTAL

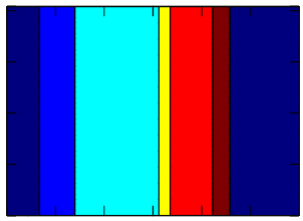
Draws horizontal bars from points to axis plane. If DATAGRID=Yes then grid lines are drawn, default=No. BARWIDTH sets relative width of rectangles, default=0.7.



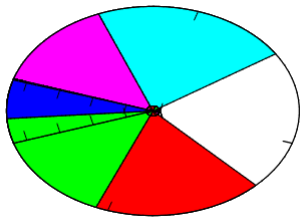
AXS_XORIGIN=0
AXS_YORIGIN=0
BOX=Yes

CHART

Draws colored stripes (boxes). If DATAGRID=Yes then black border lines are drawn, default=No. If PIECHART=Yes cylindrical coordinates are used, default=No.



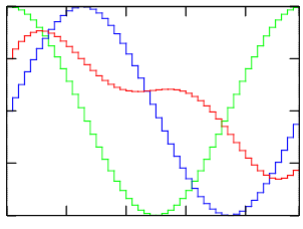
DATAGRID=Yes
BOX=Yes



DATAGRID=Yes
BOX=Yes
PIECHART=Yes
COLORSCHEME=bgr cmy

STEP

Draws continuous stairs for points to axis plane. DS_COLOR, DS_LINESTYLE and DS_LINEWIDTH are used to configure the lines. DS_COLOR, DS_MARKSTYLE and DS_MARKSIZE are used to configure the marks.

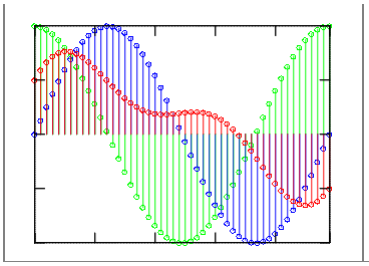


(3 plots)
BOX=Yes

STEM

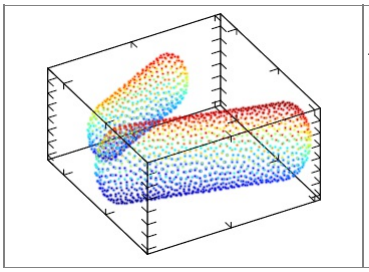
Draws vertical lines from points to axis plane. DS_COLOR, DS_LINESTYLE and DS_LINEWIDTH are used to configure the lines. DS_COLOR, DS_MARKSTYLE and DS_MARKSIZE are used to configure the marks.

(3 plots)
DS_MARKSTYLE=HOLLOW_CIRCLE
DS_SHOWMARKS=Yes
AXS_XORIGIN=0
AXS_YORIGIN=0
BOX=Yes



DOTS

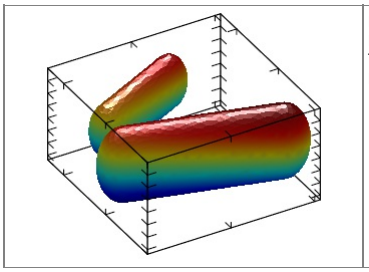
Draws arbitrary placed points. Colors will be used from the previous color scheme or from COLORSCHEME if defined.



ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes

CRUST

This will reconstruct and draw the surface for arbitrary placed points. If DATAGRID=Yes then wire plot is produced, default=No. Colors will be used from the previous color scheme or from COLORSCHEME if defined.



ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes

For Planar Datasets

For all planar modes colors will be used from the previous color scheme or from COLORSCHEME if defined. COLORSCHEME is a string that can specify a group of colors to be used by the plot.

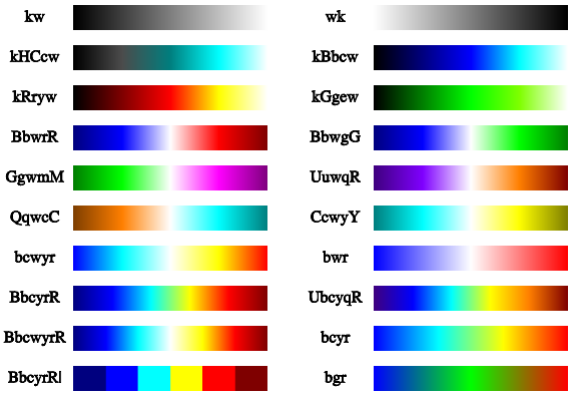
Colors in a color scheme are specified by the codes "wkrgbcymhRGBCYMHWlenupqLENUPQ" only. A brightness weight from 1 to 9 can also be used to change the default value from 5 normal, to 1 very dark, and to 9 very bright.

Also the symbol 'd' denotes the interpolation by 3D position instead of the coloring by amplitude. Symbol '|' disables color interpolation in color scheme, which can be useful, for example, for sharp colors during matrix plotting.

For coloring by amplitude (most common) the final color is a linear interpolation of color array. The color array is constructed from the string ids. The argument is the amplitude normalized based on COLORBARRANGE. When coloring by coordinate, the final color is determined by the position of the point in 3D space and is calculated from combining the first three elements of color array with the x, y and z normalizes values. This type of coloring is useful for isosurface plot where color may show the exact position of a piece of surface.

Here are some examples or color codes and color schemes:

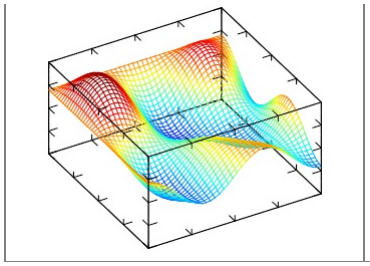
r1	r3	r5	r7	r9
b	g	r	h	w
B	G	R	H	W
c	m	y	w	p
C	M	Y	k	P
l	e	n	u	q
L	E	N	U	Q



PLANAR_MESH

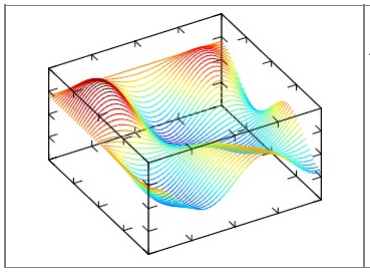
Draws mesh lines for the surface. Mesh lines are plotted for each z slice of the data. DS_LINESTYLE and DS_LINEWIDTH are used to configure the lines.

ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes



PLANAR_FALL

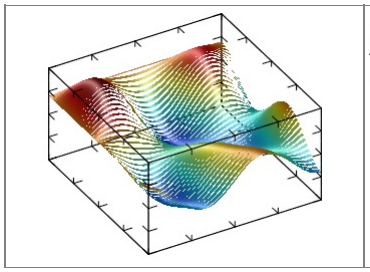
Draws fall lines for the surface. DS_LINESTYLE and DS_LINEWIDTH are used to configure the lines. If DIR=X, then lines are drawn along x-direction else lines are drawn along y-direction, default=Y.



ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes

PLANAR_BELT

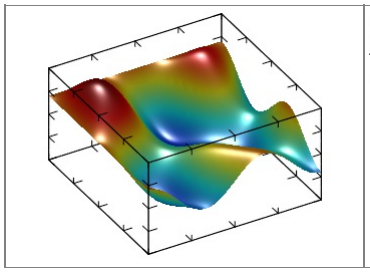
Draws belts for the surface. If DIR=X, then lines are drawn along x-direction else lines are drawn along y-direction, default=Y.



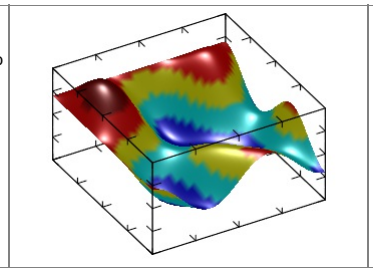
ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes

PLANAR_SURFACE

Draws the surface. If DATAGRID=Yes then grid lines are drawn, default=No.



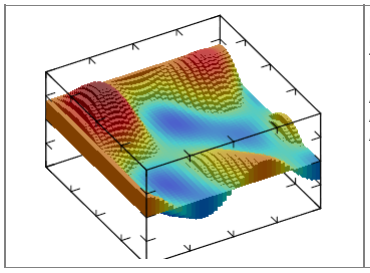
ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes



ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes
COLORSCHEME=BbcyrR|

PLANAR_BOXES

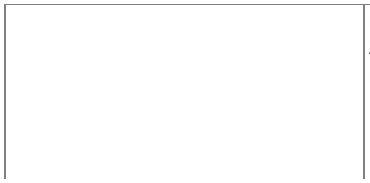
Draws vertical boxes for the surface. If DATAGRID=Yes then box lines are drawn, default=No.



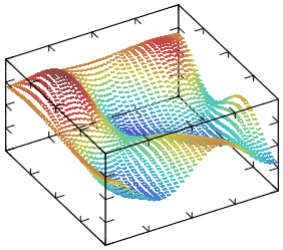
ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes
AXS_XORIGIN=0
AXS_YORIGIN=0
AXS_ZORIGIN=0

PLANAR_TILE

Draws horizontal tiles for the surface, it can be seen as 3D generalization of STEP.

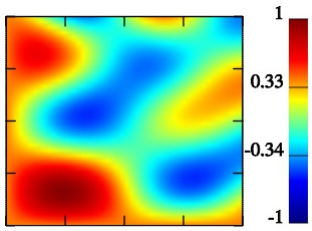


ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes



PLANAR_DENSITY

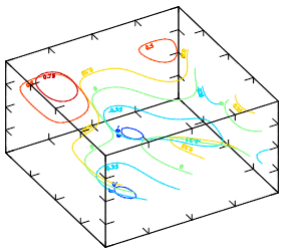
Draws density plot for the surface at minimum z coordinate.



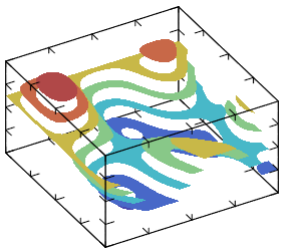
BOX=Yes
TRANSPARENT=No
COLORBAR=Yes

PLANAR_CONTOUR

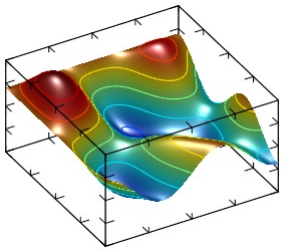
Draws contour lines for the surface at the minimum z coordinate. CONTOURCOUNT defines the number of contour lines, default=7. If CONTOURFILLED=Yes draws solid (or filled) contour lines for the surface, default=No. If CONTOURLABELS is defined then contour labels will be drawn BELLOW or ABOVE the contours.



ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes
CONTOURLABELS=BELLOW



ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes
CONTOURFILLED=Yes

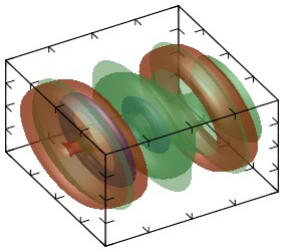


ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes

(dual plot)
DS_MODE=PLANAR_SURFACE
DS_MODE=PLANAR_CONTOUR

PLANAR_AXIALCONTOUR

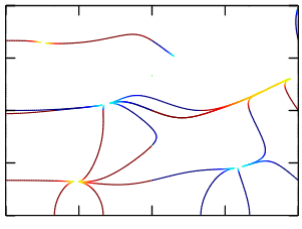
Draws a surface which is result of the contour plot rotation for the surface. AXIALCOUNT defines the number of elements distributed in the COLORBARRANGE interval, default=3.



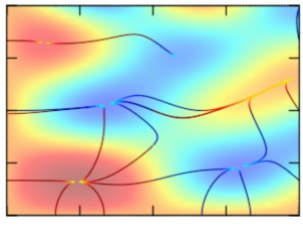
ROTATE=40:0:60
BOX=Yes
TRANSPARENT=Yes
LIGHT=Yes

PLANAR_GRADIENTLINES

Draws gradient lines for scalar field defined by the surface at minimum z coordinate. Number of lines is proportional to GRADLINESCOUNT, default=5 . If GRADLINESCOUNT<0 then lines start from borders only. Lines are plotted for each z slice of the data.



BOX=Yes
TRANSPARENT=Yes
LIGHT=Yes



BOX=Yes
TRANSPARENT=Yes
LIGHT=Yes

(dual plot)
DS_MODE=PLANAR_GRADIENTLINES
DS_MODE=PLANAR_DENSITY

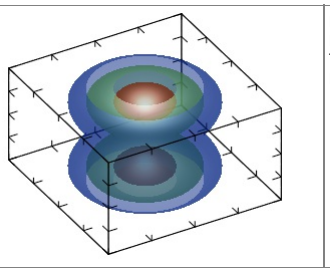
For Volumetric Datasets

For all volumetric modes colors will be used from the previous color scheme or from COLORSCHEME if defined.

VOLUME_ISOSURFACE

Draws isosurface plot for the volume. If DATAGRID=Yes then wire plot is produced, default=No. if ISOVALUE is defined only 1 isosurface is plot, else ISOCOUNT (default=3) surfaces are plot distributed in the COLORBARRANGE interval.

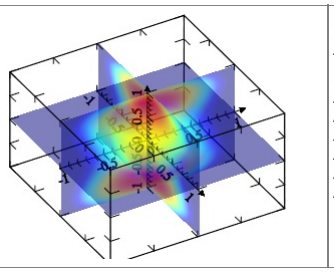
Note, that there is possibility of incorrect plotting due to uncertainty of cross-section defining if there are two or more isosurface intersections inside one cell.



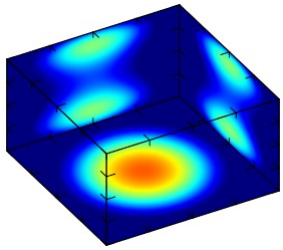
ROTATE=40:0:60
BOX=Yes
TRANSPARENT=Yes
LIGHT=Yes

VOLUME_DENSITY

Draws density plot for the volume. If DATAGRID=Yes then grid lines are drawn, default=No. If PROJECT=Yes draws density plot in x, y, or z plain, default=No. When PROJECT=Yes, PROJECTVALUEX, PROJECTVALUEY and PROJECTVALUEZ, are used to select data at the given coordinate, if they are not defined AXS_?ORIGIN is used accordingly. When PROJECT=No, SLICEX, SLICEY and SLICEZ, are used to define the slice where the plot is done, default is -1 (central). SLICEDIR defines which directions are used, default "XYZ".



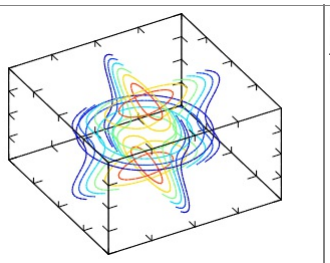
ROTATE=40:0:60
BOX=Yes
TRANSPARENT=Yes
LIGHT=No
AXS_XORIGIN=0
AXS_YORIGIN=0
AXS_ZORIGIN=0
AXS_X=Yes
AXS_Y=Yes
AXS_Z=Yes



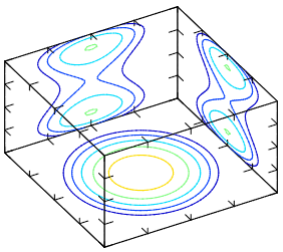
ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=No
PROJECT=Yes
PROJECTVALUEX=-1
PROJECTVALUEY=1
PROJECTVALUEZ=-1

VOLUME_CONTOUR

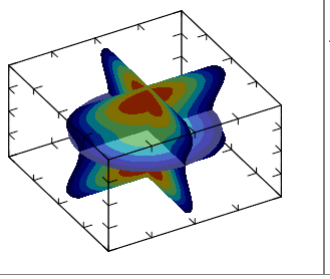
Draws contour plot for the volume. If DATAGRID=Yes then grid lines are drawn, default=No. If PROJECT=Yes draws contour plot in x, y, or z plain, default=No. When PROJECT=Yes, PROJECTVALUEX, PROJECTVALUEY and PROJECTVALUEZ, are used to select data at the given coordinate, if they are not defined AXS_?ORIGIN is used accordingly. When PROJECT=No, SLICEX, SLICEY and SLICEZ, are used to define the slice where the plot is done, default is -1 (central). SLICEDIR defines which directions are used, default "XYZ". If CONTOURFILLED=Yes draws solid (or filled) contour lines for the surface, default=No. CONTOURCOUNT defines the number of contour lines, default=7. Where lines are used, DS_LINESTYLE and DS_LINEWIDTH are used to configure the lines.



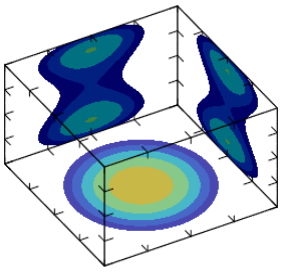
ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes



ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes
PROJECT=Yes
PROJECTVALUEX=-1
PROJECTVALUEY=1
PROJECTVALUEZ=-1



ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes

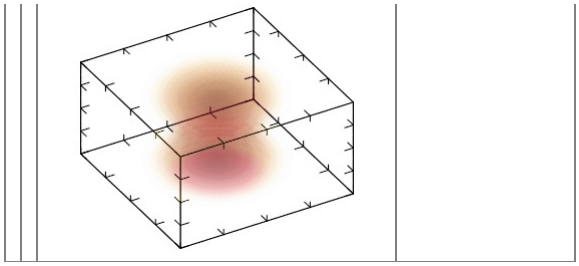


ROTATE=40:0:60
BOX=Yes
TRANSPARENT=No
LIGHT=Yes
PROJECT=Yes
PROJECTVALUEX=-1
PROJECTVALUEY=1
PROJECTVALUEZ=-1
CONTOURFILLED=Yes

VOLUME_CLOUD

Draws cloud plot for the volume. This plot is a set of cubes with color and transparency proportional to value of ALPHA. The resulting plot is like cloud – low value is transparent but higher ones are not. If CLOUDLOW=Yes then lower quality plot will be produced with much low memory usage.

ROTATE=40:0:60
BOX=Yes
TRANSPARENT=Yes
LIGHT=No
COLORSCHEME=wyrRk



IupMglLabel (since 3.11.1)

Creates a label interface element using MathGL so it can display TeX symbols. It inherits from IupMglPlot.

Creation

```
Ihandle* IupMglLabel(const char *title); [in C]
iup.mgllabel{[title = title: string]} -> (ih: ihandle) [in Lua]
mgllabel(title) [in LED]
```

title: Text to be shown on the label. It can be NULL. It will set the LABELTITLE attribute.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

The **IupMglLabel** element handle all attributes defined for a IupMglPlot control.

BGCOLOR: By default will use the background color of the native parent.

LABELTITLE (non inheritable): Label's text.

LABELFONT (non inheritable): same as **DRAWFONT**.

LABELFONTSIZE (non inheritable): same as **DRAWFONTSIZE**.

LABELFONTSTYLE (non inheritable): same as **DRAWFONTSTYLE**.

Notes

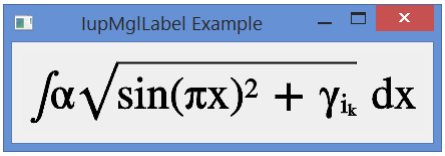
All MARGIN* attributes are set to NO. All AXS_* attributes are set to NO. EXPAND and BORDER are set to NO. And POSTDRAW_CB callback is implemented.

The **IupMglLabel** can NOT contain images.

The Natural size is NOT computed from its contents. The application must set SIZE or RASTERSIZE and then set **LABELFONTSIZE** to obtain the desired result. MathGL does not have the same font scale as IUP.

Examples

```
lbl = IupMglLabel("\\int \\alpha \\sqrt{sin(\\pi x)^2 + \\gamma_{i_k}} dx");
IupSetAttribute(lbl, "RASTERSIZE", "400x80");
IupSetAttribute(lbl, "LABELFONTSIZE", "10");
```



Browse for Example Files

See Also

IupMglPlot.

IupOleControl [Windows only]

The IupOleControl hosts an windows OLE control (also named ActiveX control), allowing it to be used inside IUP dialogs. There are many OLE controls available, like calendars, internet browsers, PDF readers etc.

Notice that IupOleControl just takes care of the visualization of the control (size and positioning), and map some callbacks (navigate and new window) using a listener interface to sink events. It does not deal with properties, methods and events. The application must deal with them using the COM interfaces offered by the control. Nevertheless, using IupLua together with LuaCOM makes it possible to use OLE controls very easily in Lua, accessing their methods, properties and events similarly to the other IUP elements.

Notice that this control works only on Windows.

When linking with GCC add the "oleaut32" and "uuid" to the list of libraries.

Initialization and usage

The **IupOleControlOpen** function must be called after a **IupOpen**, so that the control can be used. The iupole.h file must also be included in the source code. The program must be linked to the controls library (iupole).

To make the control available in Lua use require"iupluaole" or manually call the initialization function in C, **iupolelua_open**, after calling **iuplua_open**. When manually calling the function the iupluaole.h file must also be included in the source code, and the program must be linked to the lua control library (iupluaole).

Creation

```
Ihandle* IupOleControl(const char* ProgID); [in C]
iup.olecontrol{ProgID: string} -> (ih: ihandle) [in Lua]
olecontrol(ProgID) [in LED]
```

ProgID: the programmatic identifier of the OLE control. This can be found in the documentation of the OLE control or by browsing the list of registered controls, using tools like OleView.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

DESIGNMODE: activates the design mode. Some controls behave differently when in design mode. See this article for more information about design mode. Can be YES or NO. Default value: "NO".

DESIGNMODE_DONT_NOTIFY: sets the design mode, but do not notify the native control.

IUNKNOWN (read-only): Returns the IUnknown pointer for the control. This pointer is necessary to access methods and properties of the control in C/C++ code.

The control's specific attributes shall be accessed using the COM mechanism (see section below for more information).

Some IupCanvas attributes may also work, like:

ACTIVE, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, SIZE, RASTERSIZE, ZORDER, VISIBLE

Callbacks

In C/C++, the OLE control's callbacks (events, in ActiveX terms) shall be set using the control's interface and the COM mechanism. When using IupLua, it's possible to call methods, access properties and receive events from the OLE control using the LuaCOM library. When the LuaCOM library is loaded, call elem:CreateLuaCOM so a LuaCOM object is created and stored in the **"elem.com"** field of the object returned by iup.olecontrol. This LuaCOM object can be used to access properties, methods and events in a way very similar to VB. See the examples for more information.

Some **IupCanvas** callbacks may also work, like:

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB.

Additional Methods in Lua

```
ih:CreateLuaCOM()
```

If LuaCOM is loaded and the IUNKNOWN is valid then set:

```
ih.com = luacom.CreateLuaCOM(luacom.ImportIUnknown(ih.iunknown))
```

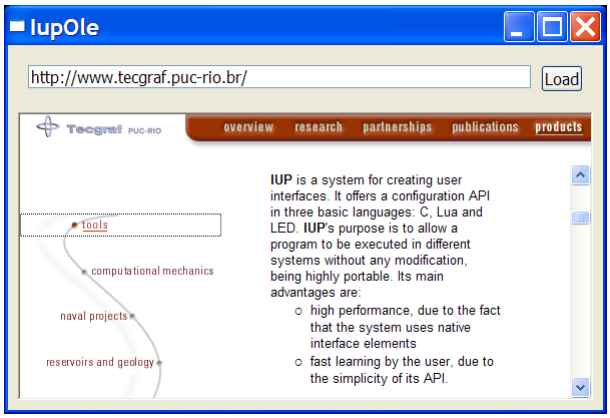
Notes

To learn more about OLE and ActiveX:

http://www.microsoft.com/com
http://www.webopedia.com/TERM/A/ActiveX_control.html
http://msdn.microsoft.com/workshop/components/activex/activex_node_entry.asp
http://activex.microsoft.com/activex/activex/

Examples

Browse for Example Files



IupScintilla (since 3.8)

Creates a multiline source code text editor that uses the Scintilla library.

Scintilla is a free library that provides text-editing functions, with an emphasis on advanced features for source code editing. It comes with complete source code and a license that permits use in any free project or commercial product, and it is available on http://www.scintilla.org/.

IupScintilla library includes the Scintilla 3.6.2 source code, so no external references are needed.

Supported in Windows and in the systems the GTK driver is supported.

Initialization and Usage

The **IupScintillaOpen** function must be called after a **IupOpen**, so that the control can be used. The "iup_scintilla.h" file must also be included in the source code. The program must be linked to the controls library (iup_scintilla), and with the "imm32.lib" library in Windows.

To make the control available in Lua use require"iuplua_scintilla" or manually call the initialization function in C, **iupscintillalua_open**, after calling **iuplua_open**. When manually calling the function the iuplua_scintilla.h file must also be included in the source code and the program must be linked to the Lua control library (iuplua_scintilla).

Creation

```
Ihandle* IupScintilla(void); [in C]
```

```
iup.scintilla{} -> (ih: ihandle) [in Lua]
scintilla(action) [in LED]
```

This function returns the identifier of the created editing component, or NULL if an error occurs.

Auxiliary Functions

```
sptr_t IupScintillaSendMessage(Ihandle* ih, unsigned int iMessage, uptr_t wParam, sptr_t lParam); [in C]
Not available in Lua.
```

Sends a message to the Scintilla control in any platform. (since 3.11)

Attributes

General

BORDER (creation only): Shows a border around the text. Default: "YES".

CANFOCUS (creation only) (non inheritable): enables the focus traversal of the control. In Windows the control will still get the focus when clicked. Default: YES.

CLIPBOARD (non inheritable): clear, cut, copy or paste the selection to or from the clipboard. Values: "CLEAR", "CUT", "COPY", "PASTE". Returns Yes or No, if data can be pasted from the clipboard.

CURSOR (non inheritable): defines the cursor type. Can be: "NORMAL" or "WAIT" (displays a wait cursor when the mouse is over or owned by the control).

DROPFILESTARGET [Windows and GTK Only] (non inheritable): Enable or disable the drop of files. Default: NO, but if DROPFILES_CB is defined when the element is mapped then it will be automatically enabled.

KEYSUNICODE [Windows Only] (non inheritable): allow processing of Unicode typed characters. Default: NO. (since 3.9)

OVERWRITE (non inheritable): turns the overwrite mode ON or OFF. When enabled, each typed character replaces the character to the right of the text caret. When disabled, characters are inserted at the caret.

READONLY (non inheritable): Allows the user only to read the contents, without changing it. Restricts the insertion using keyboard input and attributes. Navigation keys are still available. Possible values: "YES" and "NO". Default: NO.

SAVEDSTATE (non inheritable): sets the current state of the document to saved (given value is ignored), returns Yes or No if the document has been modified. After setting the SAVEDSTATE, when editing is done the SAVEPOINT_CB callback is called with status=0. When undo is performed back to the point were the saved state was set the callback is called again with status=1.

SIZE (non inheritable): Since the contents can be changed by the user, the **Natural Size** is not affected by the text contents. Use VISIBLECOLUMNS and VISIBLELINES to control the **Natural Size**.

USEPOPUP (non inheritable): allows to disable the default editing menu shown when the user clicks with the right button. Default: Yes.

VISIBLECOLUMNS: Defines the number of visible columns for the **Natural Size**, this means that will act also as minimum number of visible columns. It uses a wider character size then the one used for the SIZE attribute so strings will fit better without the need of extra columns. As for SIZE you can set to NULL after map to use it as an initial value. Default: 50.

VISIBLELINES: Defines the number of visible lines for the **Natural Size**, this means that will act also as minimum number of visible lines. As for SIZE you can set to NULL after map to use it as an initial value. Default: 10.

VISIBLELINESCOUNT (non inheritable, read-only): returns the number of actual visible lines.

WORDWRAP (non inheritable): If enabled will force a word wrap of lines that are greater than the with of the control, and the horizontal scrollbar will be removed. Default: NO.

WORDWRAPVISUALFLAGS (non inheritable): enable the drawing of visual flags to indicate a line is wrapped. Can be: MARGIN (at the line number margin), START (start of wrapped line), END (end of wrapped line) or NONE. Default: NONE.

Text Retrieval and Modification

APPEND (non inheritable, write-only): Inserts a text at the **end** of the current text. If APPENDNEWLINE=YES, a "\n" character will be automatically inserted before the appended text if the current text is not empty (APPENDNEWLINE default is YES).

CHAR*id* (non inheritable, read-only): returns the character at a given position, considering the "id" as the position.

CLEARALL (non inheritable, write-only): deletes all the text (unless the document is read-only).

COUNT (non inheritable, read-only): returns the number of characters in the text.

DELETERANGE (non inheritable, write-only): Deletes a range of text in the document. It uses a string format "**pos,len**" in order to indicate the start position and text length to delete.

INSERT*id* (non inheritable, write-only): Inserts a text string at position "id" or at the current position if pos is -1 or omitted. If the current position is after the insertion point then it is moved along with its surrounding text but no scrolling is performed.

LINE*id* (non inheritable, read-only): returns the text of the line, considering the "id" as the line number. It does not include the "\n" character. Number lines starts at 0.

LINECOUNT (non inheritable, read-only): returns the number of lines in the text.

LINEVALUE (non inheritable, read-only): returns the text of the line where the caret is. It does not include the "\n" character.

PREPEND (non inheritable, write-only): Inserts a text at the **begin** of the current text. If APPENDNEWLINE=YES, a "\n" character will be automatically inserted after the prepended text if the current text is not empty (APPENDNEWLINE default is YES).

VALUE (non inheritable): Text entered by the user. The "\n" character indicates a new line. After the element is mapped and if there is no text will return the empty string "". This replaces all the text in the document with the zero terminated text string you pass in.

Annotation

ANNOTATIONTEXT*id* (non inheritable): defines and returns an annotation displayed underneath a specific line, considering the "id" as the line number. An annotation may consist of multiple lines separated by '\n'.

ANNOTATIONSTYLE*id* (non inheritable): sets and gets a particular style to the annotation, considering the "id" as the line number.

ANNOTATIONSTYLEOFFSET (non inheritable): sets and gets a style offset, in order to separate standard text styles from annotation styles.

ANNOTATIONVISIBLE (non inheritable): enable or disable annotations. Can be "HIDDEN" (not displayed), "STANDARD" (displayed) or "BOXED" (displayed and surrounded by a box). Default HIDDEN.

ANNOTATIONCLEARALL (non inheritable, write-only): deletes all annotations.

Auto-Completion (since 3.10)

AUTOCSHOW*id* (non inheritable, write only): causes a list of words to be displayed. The words are separated by a space. "id" defines the number of characters of the word already entered by user.

AUTOCCANCEL (non inheritable, write only): cancels any displayed auto-completion list. When in auto-completion mode, the list should disappear when the user types a character that can not be part of the auto-completion.

AUTOCACTIVE (non inheritable, read only): returns YES if there is an active auto-completion list and NO if there is not.

AUTOCPOSSTART (non inheritable, read only): returns the current position when the list of words started to be shown.

AUTOCCOMPLETE (non inheritable, write only): triggers auto-completion. This has the same effect as the tab key.

AUTOCSELECT (non inheritable, write only): selects an item in the auto-completion list. It searches in the list of words for the first that matches of value (comparisons are case sensitive). If the item is not found, no item is selected.

AUTOCSELECTEDINDEX (non inheritable, read only): retrieves the current selection index, set by AUTOCSELECT attribute.

AUTOCDROPRESTOFWORD (non inheritable): when an item is selected, any word characters following the caret are first erased if this attribute is set YES. The default is NO.

AUTOCMAXHEIGHT (non inheritable): sets and gets the maximum number of rows that will be visible in an auto-completion list. If there are more rows in the list, then a vertical scrollbar is shown. The default is 5.

AUTOCMAXWIDTH (non inheritable): the maximum width of an auto-completion list expressed as the number of characters in the longest item that will be totally visible. The default is 0 (in this case, the list width is calculated to fit the item with the most characters).

Brace Highlighting

BRACEHIGHLIGHT (non inheritable, write only): highlights the brace, defined by its initial and final positions (format: "**pos1**:**pos2**"). Up to two characters can be highlighted in a 'brace highlighting style', which is defined as style number (See Style Definition, id = 34).

BRACEBADLIGHT (non inheritable, write only): highlights the non matching brace, based on a position. If there is no matching brace then the brace badlighting style (See Style Definition, id = 35) can be used to show the brace that is unmatched. Set -1 as position removes the highlight.

BRACEMATCH*id* (non inheritable, read only): finds a corresponding matching brace given id, the position of one brace. The brace characters handled are '(', ')', '[', ']', '{', '}', '<', and '>'. If the character at position is not a brace character, or a matching brace cannot be found, the return value is -1.

Caret and Selection

CARET (non inheritable): Position of the insertion point. The first position, **lin** or **col**, is "0". It uses a string format "**lin**,**col**" in order to indicate the caret position, where **lin** and **col** must be integer numbers.

When **lin** is greater than the number of lines, the caret is placed at the last line. When **col** is greater than the number of characters in the given line, the caret is placed after the last character of the line.

If the caret is not visible the text is scrolled to make it visible.

CARETPOS (non inheritable): Position of the insertion point using a zero based character unique index "pos". Useful for indexing the VALUE string. This removes any selection, sets the caret at pos and scrolls the view to make the caret visible, if necessary.

CARETTOVIEW (non inheritable, write only): Moves the caret to the nearest visible line. Any selection is lost.

CARETCOLOR (non inheritable): color of the caret. Values in RGB format ("r g b"). (since 3.17)

CARETSTYLE (non inheritable): style of the caret. Can be LINE, BLOCK or INVISIBLE. Default: LINE. (since 3.17)

CARETWIDTH (non inheritable): with of the caret line. Can be 0, 1, 2 or 3 pixels. Default: 1. Works only when CARETSTYLE=LINE. A size of 0 will make the caret invisible also. (since 3.17)

SELECTEDTEXT (non inheritable): Selection text. Returns NULL if there is no selection. When changed replaces the current selection. Similar to INSERT, but does nothing if there is no selection.

SELECTION (non inheritable): Selection interval. Returns NULL if there is no selection. The first position, **lin** or **col**, is "0". The accepted format is represented by the string "**lin1**,**col1**:**lin2**,**col2**", where **lin1**, **col1**, **lin2** and **col2** are integer numbers corresponding to the selection's interval. **col2** correspond to the character after the last selected character. The values ALL and NONE are also accepted.

SELECTIONPOS (non inheritable): Same as SELECTION but using a zero based character index "**pos1**:**pos2**". Useful for indexing the VALUE string. The values ALL and NONE are also accepted.

Folding

FOLDFLAGS (non inheritable, write-only): determines how folding lines are drawn. Can be: "LEVELNUMBERS", "LINEBEFORE_EXPANDED", "LINEBEFORE_CONTRACTED", "LINEAFTER_EXPANDED" or "LINEAFTER_CONTRACTED " (default).

FOLDLEVEL*id* (non inheritable): sets and gets the fold level of a "id" line. Can be: "WHITEFLAG", "HEADERFLAG", "NUMBERMASK" or "BASE" (default). If you use a Lexer, it is not recommend to set the fold level (this is far better handled by the Lexer). By contrast, get fold level is useful to decide how to handle user folding requests.

FOLDTOGGLE (non inheritable, write-only): Determines if the fold point (line number) may be either expanded, displaying all its child lines, or contracted, hiding all the child lines.

Lexer

KEYWORD*id* (non inheritable, write-only): keyword list used by the current Lexer. Until 9 lists of keywords can set up using id from 0 to 8. The value is a list of keywords separated by spaces, tabs, "\n" or "\r" or any combination of these.

KEYWORDSETS (non inheritable, read only): returns a description of all of the keyword sets separated by "\n".

LEXERLANGUAGE (non inheritable): associate the Lexer language name. It is case sensitive. Default: not defined. Set to NULL to clear the association. Can be: any name supported by Scintilla. For instance: asm, bash, freebasic, cmake, COBOL, cpp (C++), css, d, diff, eiffel, fortran, hypertext (HTML), xml, lisp, lua (Lua), makefile, matlab, mysql, nsis, pascal, perl, python, ruby, smalltalk, sql, tcl, tex, vb (Visual Basic), and many others.

LOADLEXERLIBRARY (non inheritable, write-only): Load a Lexer implemented in a dynamic library given the library file name. This is a .so file on GTK+/Linux or a .DLL file on Windows. (since 3.11)

PROPERTY (non inheritable): sets and gets Lexer properties using "name=value" string pairs, where name is case sensitive and value is the associated string. There is no limit to the number of keyword pairs you can set, other than available memory. To retrieve a property first set the PROPERTYNAME attribute, the PROPERTY attribute will return its value.

PROPERTYNAMES (non inheritable, read only): returns a list of property names separated by "\n". If the Lexer does not support this information then an empty string is returned.

Margins

MARGINMASKFOLDERS*id* (non inheritable): defines if a margin is folding or non-folding. Can be: Yes or No.

MARGINSENSITIVE*id* (non inheritable): determines if a margin is sensitive or not. Margins that are not sensitive act as selection margins which make it easy to select ranges of lines. By default, all margins are insensitive. Can be: YES or NO.

MARGINTYPE*id* (non inheritable): set and get the type of a margin. The margin argument should be 0, 1, 2, 3 or 4. Each margin can be set to display only symbols, line numbers, or text. You can use the predefined values "SYMBOL", "NUMBER", "TEXT", "RTEXT" (right justify text), "BACKGROUND" or "FOREGROUND" (the latter two used for symbol margins that set their background or foreground using the style default colors).

MARGINWIDTH*id* (non inheritable): width of a margin in pixels (Default value: 0). A margin with width=0 is invisible. The margins are numbered 0 to 4. By default, Scintilla margin 0 is set to display line numbers, margin 1 is set to display non-folding symbols, and margin 2 is set to display folding symbols. Margins 3 and 4 are free for user default. However, you can set the margins to be whatever you wish using MARGINTYPEid.

MARGINLEFT (non inheritable): size of the blank margin on the left side. Default: 1.

MARGINRIGHT (non inheritable): size of the blank margin on the right side. Default: 1.

MARGINTEXT*id* (non inheritable): controls the text of each line of a text margin. id is the line number.

MARGINTEXTSTYLE*id* (non inheritable): controls the style of the text of each line of a text margin. id is the line number.

MARGINTEXTCLEARALL (non inheritable, write-only): clear all text and styles of a text margin.

MARGINCURSOR*id* (non inheritable): set and get the arrow cursor normally shown over margins. Can be: "REVERSEARROW" (default) or "ARROW".

Markers

MARKERDEFINE (non inheritable, write-only): Defines a marker using its number and its symbol. Format: "number=symbol".

> **Marker numbers**: can be a number in the range 0 to 31, or the pre-defined names: "FOLDEREND", "FOLDEROPENMID", "FOLDERMIDTAIL", "FOLDERSUB", "FOLDER" and "FOLDEROPEN".
> **Marker symbols:** (see MARKERSYMBOL).

MARKERSYMBOL*id* (non inheritable): Defines a marker symbol given its number. id can be from 0 to 31.

> **Marker symbols:** "CIRCLE", "ROUNDRECT", "ARROW", "SMALLRECT", "SHORTARROW", "EMPTY", "ARROWDOWN", "MINUS", "PLUS", "VLINE", "LCORNER", "TCORNER", "BOXPLUS", "BOXPLUSCONNECTED", "BOXMINUS", "BOXMINUSCONNECTED", "LCORNERCURVE", "TCORNERCURVE", "CIRCLEPLUS", "CIRCLEPLUSCONNECTED", "CIRCLEMINUS", "CIRCLEMINUSCONNECTED", "BACKGROUND" (line background color), "DOTDOTDOT", "ARROWS", "FULLRECT" (margin background color), "LEFTRECT", "UNDERLINE" (underline across the line), "RGBAIMAGE" and "CHARACTER+c" (where c is an ASCii character code).

MARKERFGCOLOR*id* (non inheritable, write only): defines the foreground color of a marker number (id). Values in RGB format ("r g b").

MARKERBGCOLOR*id* (non inheritable, write only): defines the background color of a marker number (id). Values in RGB format ("r g b").

MARKERBGCOLORSEL*id* (non inheritable, write only): defines the highlight background colour of a marker number (id) when its folding block is selected. Values in RGB format ("r g b").

MARKERALPHA*id* (non inheritable, write only): defines the alpha value of a marker number (id). Markers may be drawn translucently when there are no margins.

MARKERRGBAIMAGE*id* (non inheritable, write only): defines the image name to be used on a marker number. Use IupSetHandle or IupSetAttributeHandle to associate an image to a name. See also IupImage. It must be an image created with the **IupImageRGBA** constructor, it can not be a image loaded from stock or resources.

MARKERRGBAIMAGESCALE (non inheritable, write only): defines the image scale factor, in percent (1-100).

MARKERHIGHLIGHT (non inheritable): enable or disable the the highlight folding block when it is selected. (i.e. block that contains the caret). Can be Yes or No. Default: No.

MARKERADD*id* (non inheritable, write-only): adds marker number to a line (id). Internally, sets the marker handle number (LASTMARKERADDHANDLE attribute) that identifies the added marker (or -1 for invalid line and out of memory), which may be useful to find where a marker is after moving or combining lines removes markers of the given number from all lines. If marker number is -1, it deletes all markers from all lines.

MARKERDELETE*id* (non inheritable, write-only): deletes marker number given a line number (id). If marker number is -1, all markers are deleted from the line.

MARKERDELETEALL (non inheritable, write-only): removes markers of the given number from all lines. If marker number is -1, it deletes all markers from all lines.

MARKERGET*id* (non inheritable, read-only): returns a 32-bit integer that indicates which markers were present on the line (id). Bit 0 is set if marker 0 is present, bit 1 for marker 1 and so on.

MARKERNEXT*id* (non inheritable, write-only): searches a given marker number, starting at line number (id) and continuing forwards to the end of the file. Internally, sets the the line number of the first line that contains the marker (LASTMARKERFOUND attribute) or -1, if no marker is found.

MARKERPREVIOUS*id* (non inheritable, write-only): searches a given marker number, starting at line number (id) and continuing backwards to the start of the file. Internally, sets the the line number of the first line that contains the marker (LASTMARKERFOUND attribute) or -1, if no marker is found.

MARKERLINEFROMHANDLE*id* (non inheritable, read-only): searches the document for the marker given its handle returned in MARKERADDid (use the LASTMARKERADDHANDLE as attribute value) and returns the line number of the first line that contains the marker or -1, if no marker is found.

MARKERDELETEHANDLE (non inheritable, write-only): searches the document for the marker with this handle (use the LASTMARKERADDHANDLE as attribute value) and deletes the marker if it is found.

LASTMARKERADDHANDLE (non inheritable, read-only): returns the last marker handle set by the MARKERADDid attribute.

LASTMARKERFOUND (non inheritable, read-only): returns the last line number that contains a marker found by the MARKERNEXTid, MARKERPREVIOUSid or MARKERLINEFROMHANDLE attributes.

Scrolling

SCROLLBAR (creation only): Associates an automatic horizontal and/or vertical scrollbar. Can be: "VERTICAL", "HORIZONTAL", "YES" (both) or "NO" (none). Default: "YES". For all systems, when SCROLLBAR is NO, the natural size will always include its size even if the native system hides the scrollbar.

SCROLLBY (non inheritable, write only): Scroll the text by the given offsets in the format "lin,col". Positive lin values increase the line number at the top of the screen (i.e. they move the text upwards). Positive col values increase the column at the left edge of the view (i.e. they move the text leftwards). (since 3.17)

SCROLLTOCARET (non inheritable, write only): Scroll the text to make the caret position visible.

SCROLLWIDTH (non inheritable): controls the document width in pixels. Default: 2000.

Search and Replace (since 3.10)

SEARCHINTARGET (non inheritable, write only): This searches for the first occurrence of a text string in the target defined by TARGETSTART and TARGETEND. If the search succeeds, the target is set to the found text.

SEARCHFLAGS (non inheritable): sets and gets the search flags used in SEARCHINTARGET attribute. Possible values: MATCHCASE, WHOLEWORD, WORDSTART, REGEXP and POSIX. The flag options are combined using "|" as separators. Use NULL to reset all flags.

TARGETSTART (non inheritable): sets and gets the start of target. When searching in non-regular expression mode, you can set TARGETSTART greater than TARGETEND to find the last matching text in the target rather than the first matching text. If set 0, target start will be 1 (first position of text).

TARGETEND (non inheritable): sets and gets the end of target. If set 0, target end will be the last position of text.

TARGETFROMSELECTION (non inheritable, write only): set the target start and end from current position of the selection.

REPLACETARGET (non inheritable, write only): replaces the target text. After replacement, the target range refers to the replacement text.

Style Definition (See Style Definition)

BGCOLOR: Background color of the text. Default: the global attribute TXTBGCOLOR. If changed it will affect the default style.

FGCOLOR: Text color. Default: the global attribute TXTFGCOLOR. If changed it will affect the default style.

FONT: the text font. Default: the global attribute DEFAULTFONT. If changed it will affect the default style.

STYLEBGCOLOR*id* (non inheritable): background color for a style (See Style Definition). Values in RGB format ("r g b").

STYLEBOLD*id* (non inheritable): the boldness of a font (See Style Definition).

STYLECASE*id* (non inheritable): determines how text is displayed (See Style Definition). Values: LOWERCASE, UPPERCASE or MIXED (default).

STYLECHARSET*id* (non inheritable): sets and gets a style to use a different character set than the default (See Style Definition). Can be ANSI (default), EASTEUROPE, RUSSIAN, GB2312, HANGUL or SHIFTJIS.

STYLECLEARALL (non inheritable): sets all styles to have the same attributes as default global style (id = 32) (See Style Definition).

STYLEEOLFILLED*id* (non inheritable): If the last character in the line has a style with this attribute set, the remainder of the line up to the right edge of the window is filled with the background color set for the last character (See Style Definition). Can be YES (italic) or NO.

STYLEFGCOLOR*id* (non inheritable): foreground color for a style (See Style Definition). Values in RGB format ("r g b").

STYLEFONT*id* (non inheritable): sets and gets the font name (See Style Definition). Scintilla caches fonts by their names, but the cache is case sensitive.

STYLEFONTSIZE*id* (non inheritable): sets and gets the font size (See Style Definition), using a integer number of points.

STYLEFONTSIZEFRAC*id* (non inheritable): sets and gets the font size (See Style Definition), using a fractional point size in hundredths of a point. For example, a text size of 9.4 points is set with value = 940.

STYLEHOTSPOT*id* (non inheritable): used to mark ranges of text that can detect mouse clicks (See Style Definition). The cursor changes to a hand over hotspots, and the foreground, and background colors may change and an underline appear to indicate that these areas are sensitive to clicking. This may be used to allow hyperlinks to other documents.

STYLEITALIC*id* (non inheritable): the italic style of a font (See Style Definition). Can be YES (italic) or NO.

STYLERESET (non inheritable, write-only): Resets to the initial Scintilla style default (See Style Definition).

STYLEUNDERLINE*id* (non inheritable): determines if the underline is drawn, using a foreground color (See Style Definition). Can be YES (underline) or NO.

STYLEVISIBLE*id* (non inheritable): determines if the text is visible (YES) or hidden (NO) (See Style Definition).

STYLEWEIGHT*id* (non inheritable): the weight of a font (See Style Definition). Predefined values: NORMAL, SEMIBOLD and BOLD. The weight can also be a number between 1 and 999 with 1 being very light and 999 very heavy.

Styling

CLEARDOCUMENTSTYLE (non inheritable, write-only): clear all styling information and reset the folding state.

STARTSTYLING (non inheritable, write only): prepares for styling by setting the styling position.

STYLING*id* (non inheritable, write only): sets the style of given length characters starting at the styling position and then increases the styling position by length. id is the style.

Tabs and Indentation Guides

TABSIZE (non inheritable): Controls the number of characters for a tab stop. Default: 8.

INDENTATIONGUIDES (non inheritable): dotted vertical lines that appear within indentation white space every indent size columns. Can be: NONE, REAL, LOOKFORWARD, LOOKBOTH. Default: NONE.

HIGHLIGHTGUIDE (non inheritable): Highlights the indentation guide of a given column. When brace highlighting occurs, the indentation guide corresponding to the braces may be highlighted with the brace highlighting style (See Style Definition, id = 34). Set column to 0 to cancel this highlight.

USETABS (non inheritable): Use tabs also for indentation or only spaces. Can be Yes or No. Default: Yes.

Undo and Redo

REDO (non inheritable): redo the last operation if set to Yes, clears the undo information otherwise. Returns Yes or No if redo can be performed.

UNDO (non inheritable): undo the last operation if set to Yes, clears the undo information otherwise. Returns Yes or No if undo can be performed.

UNDOCOLLECT (non inheritable): enable or disable the undo collect of information. Can be Yes or No. Default: Yes.

White Space

EXTRAASCENT (non inheritable): sets and gets the space to be added to the maximum ascent, in order to allow for more space between lines. Default: 0.

EXTRADESCENT (non inheritable): sets and gets the space to be added to the maximum descent, in order to allow for more space between lines. Default: 0.

WHITESPACEVIEW (non inheritable): sets and gets the white space display mode. The white spaces can be: "INVISIBLE" (shown as an empty background color), "VISIBLEALWAYS" (drawn as dots and arrows) or "VISIBLEAFTERINDENT" (white space used for indentation is displayed normally but after the first visible character, it is shown as dots and arrows). Default: INVISIBLE.

WHITESPACESIZE (non inheritable): sets and gets the size of the dots used for mark space characters. Default: 1.

WHITESPACEFGCOLOR (non inheritable, write only): defines the foreground color of visible white space. Values in RGB format ("r g b"). By default the color will be defined by the Lexer, but defining this attribute will overriding the Lexer definition. Set to NULL to reset the definition and use the Lexer again.

WHITESPACEBGCOLOR (non inheritable, write only): defines the background color of visible white space. Values in RGB format ("r g b"). By default the color will be defined by the Lexer, but defining this attribute will overriding the Lexer definition. Set to NULL to reset the definition and use the Lexer again.

Zooming

ZOOMIN (non inheritable, write only): increases the zoom factor by one point if the current zoom factor is less than 20 points.

ZOOMOUT (non inheritable, write only): decreases the zoom factor by one point if the current zoom factor is greater than -10 points.

ZOOM (non inheritable): sets and gets the zoom factor directly. Limits: -10 points to zoom out and 20 points to zoom in.

ACTIVE, EXPAND, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Drag & Drop attributes are supported. See Notes bellow.

Callbacks

ACTION: Action generated when the text is edited, but before its value is actually changed. Can be generated when using the keyboard, undo/redo system or from the clipboard.

```
int function(Ihandle *ih, int insert, int pos, int length, char* text ); [in C]
ih:action(insert, pos, length: number, text: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
insert: =1 when text is inserted, =0 when text is deleted.
pos: 0 based character position when change started.
length: size of the change.

text: the inserted text value. It is NULL when insert=0.

AUTOCSELECTION_CB: Action generated when the user has selected an item in an auto-completion list. It is sent before the selection is inserted. Automatic insertion can be cancelled by setting the **AUTOCCANCEL** attribute before returning from the callback. (since 3.10.1)

```
int function(Ihandle *ih, int pos, char* text ); [in C]
ih:autocselection_cb(pos: number, text: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
pos: 0 based character start position of the word being completed.
text: the text of the selection.

AUTOCCANCELLED_CB: Called after the user has cancelled an auto-completion list. (since 3.10.1)

```
int function(Ihandle *ih); [in C]
ih:autoccancelled_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

AUTOCCHARDELETED_CB: Called after the user deleted a character while auto-completion list was active. (since 3.10.1)

```
int function(Ihandle *ih); [in C]
ih:autocchardeleted_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

BUTTON_CB: Action generated when any mouse button is pressed or released. Use IupConvertXYToPos to convert (x,y) coordinates in character positioning.

CARET_CB: Action generated when the caret/cursor position is changed.

```
int function(Ihandle *ih, int lin, int col, int pos); [in C]
ih:caret_cb(lin, col, pos: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
lin, col: line and column number (start at 0).
pos: 0 based character position.

DROPFILES_CB: Action generated when one or more files are dropped in the element.

HOTSPOTCLICK_CB: Action generated when the user clicks or double clicks on text that is in a style with the hotspot attribute set.

```
int function(Ihandle *ih, int pos, int lin, int col, char* status); [in C]
ih:hotspotclick_cb(pos, lin, col: number, status: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
pos: the character position in the document that corresponds to the hotspot click.
lin: line in the document that corresponds to the hotspot click.
col: column in the document that corresponds to the hotspot click.
status: status of mouse buttons and certain keyboard keys at the moment the event was generated. The same macros used for BUTTON_CB can be used for this status.

MARGINCLICK_CB: Action generated when the mouse button is clicked inside a margin that is marked as sensitive.

```
int function(Ihandle *ih, int margin, int lin, char* status); [in C]
ih:marginclick_cb(margin, lin: number, status: string) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
margin: the margin number that was clicked.
lin: line in the document that corresponds to the margin click.
status: status of mouse buttons and certain keyboard keys at the moment the event was generated. The same macros used for BUTTON_CB can be used for this status.

MOTION_CB: Action generated when the mouse is moved. Use IupConvertXYToPos to convert (x,y) coordinates in character positioning.

SAVEPOINT_CB: Notifies the application that a save point was reached (1) or left (0). Can be used to controls whether to display a saved or modified document. To set the save point use the SAVEDSTATE attribute.

```
int function(Ihandle *ih, int status); [in C]
ih:savepoint_cb(status: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
status: can be 1 (reached) or 0 (left).

VALUECHANGED_CB: Called after the value was interactively changed by the user.

```
int function(Ihandle *ih); [in C]
ih:valuechanged_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

ZOOM_CB: Notifies the application when the user zooms the display using the keyboard or the ZOOM attribute. Can be used to recalculate positions, such as the width of the line number margin to maintain sizes in terms of characters rather than pixels.

```
int function(Ihandle *ih, int zoomInPoints); [in C]
ih:zoom_cb(zoomInPoints: number) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
zoomInPoints: the current zoom factor.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Drag & Drop callbacks are supported. See Notes bellow.

Auxiliary Functions

IupText auxiliary functions **IupTextConvertLinColToPos** and **IupTextConvertPosToLinCol** can also be used. But here lin and col starts at 0, pos starts at 0.

Style Definition

Scintilla can automatically format the text using the Lexer or the application can manually format the text. In both cases only 256 different styles are available. Styles are numbered from 0 to 255, invalid IDs are simply ignored. When the id is not specified for an attributes the style 0 is assumed.

Each Lexer will use the available styles with its own rules, but several Lexers share many ids. Notice that language keywords and styles definitions are not pre-defined internally, the application must define them.

ID	Global styles
32	This style defines the attributes that all styles receive when the STYLECLEARALL attribute is used.
33	This style sets the attributes of the text used to display line numbers in a line number margin.
34	This style sets the attributes used when highlighting braces with the BRACEHIGHLIGHT and HIGHLIGHTGUIDE attributes.
35	This style sets the attributes used when marking an unmatched brace with the BRACEBADLIGHT attribute.
36	This style sets the font used when drawing control characters.
37	This style sets the foreground and background colors used when drawing the indentation guides. Used when INDENTATIONGUIDES!=NONE.

Here are some known styles for C++ and Lua:

ID	C++ styles	Lua styles
0	Default style	Default style
1	C comment	Lua comment
2	C++ comment line	Lua comment line
3	JavaDoc/ Doxygen style C comment	JavaDoc/ Doxygen style Lua comment
4	Number	Number
5	Keyword	Keyword
6	String	String
7	Character	Character
8	IDL UUID	Literal string
9	Preprocessor block	Preprocessor block
10	Operator	Operator
11	Identifier	Identifier
12	End of a line where a string is not closed	End of a line where a string is not closed
13	C# verbatim string	Keyword set number 2
14	Regular expression	Keyword set number 3
15	JavaDoc/Doxygen style C++ comment line	Keyword set number 4
16	Keyword set number 2	Keyword set number 5
17	JavaDoc/Doxygen keyword	Keyword set number 6
18	JavaDoc/Doxygen keyword error	Keyword set number 7
19	Global class or typedef defined in keyword	Keyword set number 8
20	C++ raw string	Label
21	F# triple-quoted verbatim strings	
22	Hash-quoted strings	
23	Preprocessor block comment	

Notes

Enter key will add a new line, and the Tab key will insert a Tab.

Internal Drag&Drop support is enabled by default.

IupScintilla uses attributes and callbacks very similar to the **IupText** control, except for text formatting.

Although the **IupScintilla** documentation should be sufficient for most uses, some advanced features will be better understood if the Scintilla Documentation is consulted. Also some Scintilla features are not available in **IupScintilla**, so by consulting that documentation you will be able to check which one and if necessary you can request the implementation in **IupScintilla**.

Navigation, Selection and Clipboard Keys

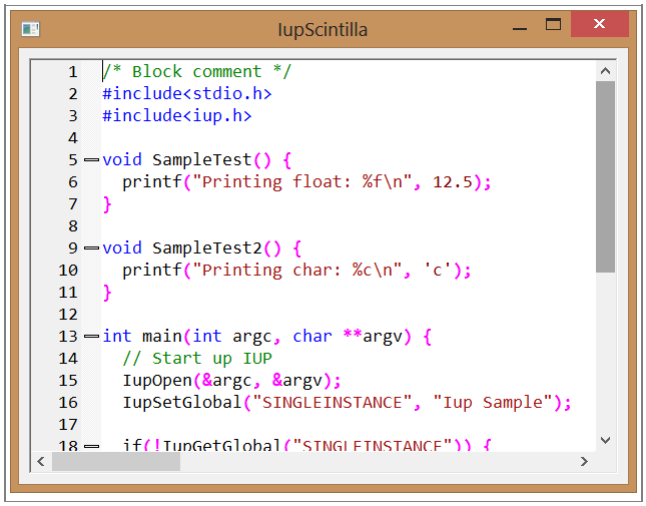
Here is a list of the common keys for all drivers. Other keys are available depending on the driver.

Keys	Action
Navigation	
Arrows	move by individual characters/lines
Ctrl+Arrows	move by words/paragraphs
Home/End	move to begin/end line
Ctrl+Home/End	move to begin/end text
PgUp/PgDn	move vertically by pages
Ctrl+PgUp/PgDn	move horizontally by pages
Selection	
Shift+Arrows	select characters
Ctrl+A	select all
Deleting	
Del	delete the character at right
Backspace	delete the character at left
Clipboard	
Ctrl+C	copy
Ctrl+X	cut
Ctrl+V	paste

Examples

Browse for Example Files

```
 IupSetAttribute(ih, "LEXERLANGUAGE", "cpp");

IupSetAttribute(ih, "KEYWORDS0", "void struct union enum char short int long double float signed unsigned const static extern auto register volatile bool class private pro
                                 "if else switch case default break goto return for while do continue typedef sizeof NULL new delete throw try catch namespace operator th
                                 "typeid and and_eq bitand bitor compl not not_eq or or_eq xor xor_eq");

//IupSetAttribute(ih, "STYLEFONT32", "Courier New");
IupSetAttribute(ih, "STYLEFONT32", "Consolas");
IupSetAttribute(ih, "STYLEFONTSIZE32", "11");
IupSetAttribute(ih, "STYLECLEARALL", "Yes");  /* sets all styles to have the same attributes as 32 */

IupSetAttribute(ih, "STYLEFGCOLOR1", "0 128 0");    // 1-C comment
IupSetAttribute(ih, "STYLEFGCOLOR2", "0 128 0");    // 2-C++ comment line
IupSetAttribute(ih, "STYLEFGCOLOR4", "128 0 0");    // 4-Number
IupSetAttribute(ih, "STYLEFGCOLOR5", "0 0 255");    // 5-Keyword
IupSetAttribute(ih, "STYLEFGCOLOR6", "160 20 20");  // 6-String
IupSetAttribute(ih, "STYLEFGCOLOR7", "128 0 0");    // 7-Character
IupSetAttribute(ih, "STYLEFGCOLOR9", "0 0 255");    // 9-Preprocessor block
IupSetAttribute(ih, "STYLEFGCOLOR10", "255 0 255"); // 10-Operator
IupSetAttribute(ih, "STYLEBOLD10", "YES");

IupSetAttribute(ih, "STYLEHOTSPOT6", "YES");

IupSetAttribute(ih, "MARGINWIDTH0", "50");

IupSetAttribute(ih, "PROPERTY", "fold=1");
IupSetAttribute(ih, "PROPERTY", "fold.compact=0");
IupSetAttribute(ih, "PROPERTY", "fold.comment=1");
IupSetAttribute(ih, "PROPERTY", "fold.preprocessor=1");

IupSetAttribute(ih, "MARGINWIDTH1", "20");
IupSetAttribute(ih, "MARGINTYPE1",  "SYMBOL");
IupSetAttribute(ih, "MARGINMASKFOLDERS1",  "Yes");

IupSetAttribute(ih, "MARKERDEFINE", "FOLDER=PLUS");
IupSetAttribute(ih, "MARKERDEFINE", "FOLDEROPEN=MINUS");
IupSetAttribute(ih, "MARKERDEFINE", "FOLDEREND=EMPTY");
IupSetAttribute(ih, "MARKERDEFINE", "FOLDERMIDTAIL=EMPTY");
IupSetAttribute(ih, "MARKERDEFINE", "FOLDEROPENMID=EMPTY");
IupSetAttribute(ih, "MARKERDEFINE", "FOLDERSUB=EMPTY");
IupSetAttribute(ih, "MARKERDEFINE", "FOLDERTAIL=EMPTY");

IupSetAttribute(ih, "FOLDFLAGS", "LINEAFTER_CONTRACTED");

IupSetAttribute(ih, "MARGINSENSITIVE1", "YES");
```



See Also

IupText, Scintilla

IupWebBrowser [GTK and Windows only] (since 3.3)

Creates a web browser control. It is responsible for managing the drawing of the web browser content and forwarding of its events.

In Linux, the implementation uses the WebKit/GTK+, the new GTK+ port of the WebKit, an open-source web content engine. More information about WebKit/GTK+ (building, dependencies, releases, etc) can be seen in Notes section. It is only available for Linux26g4 and Linux26g4_64 systems.

In Windows, the implementation uses the **IupOleControl** to embed an instance of the Internet Explorer WebBrowser control. A listener interface is used to capture and handle events using the Active Template Library (ATL) classes. More information about ATL can be seen in Notes section. So it is only available for Visual C++ compilers.

Initialization and usage

The **IupWebBrowserOpen** function must be called after **IupOpen**. The iupweb.h file must also be included in the source code. The program must be linked to the controls library (iupweb). If static linking is used then in Windows must be linked with the "iupole" library and in Linux must be linked with the "webkit-1.0" library

To make the control available in Lua use require"iupluaweb" or manually call the initialization function in C, **iupweblua_open**, after calling **iuplua_open**. When manually calling the function the iupluaweb.h file must also be included in the source code, and the program must be linked to the lua control library (iupluaweb).

Creation

```
Ihandle* IupWebBrowser(void); [in C]
iup.webbrowser{} -> (ih: ihandle) [in Lua]
webbrowser() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BACKCOUNT [GTK Only] (read only): gets the number of items that precede the current page.

BACKFORWARD (write only): sets the number of steps away from the current page and loads the history item. Negative values represent steps backward while positive values represent steps forward.

COPY (write only): copy the selection to the clipboard. (since 3.10)

FORWARDCOUNT [GTK Only] (read only): gets the number of items that succeed the current page.

HTML (write only): loads a given HTML content.

ITEMHISTORYid [GTK Only] (read only): Returns the URL associated with a specific history item. Negative "id" value represents a backward item while positive "id" value represents a forward item ("0" represents the current item).

PRINT (write only): shows the print dialog. (since 3.10)

RELOAD (write only): reloads the page in the webbrowser.

SELECTALL (write only): selects all contents. (since 3.10)

STATUS (read only): returns the load status. Can be "LOADING", "COMPLETED" or "FAILED".

STOP (write only): stops any ongoing load in the webbrowser.

VALUE: sets a specified URL to load into the webbrowser, or retrieve the current URL.

ZOOM: the zoom factor of the browser in percent. No zoom is 100%. (since 3.10)

ACTIVE, FONT, EXPAND, SCREENPOSITION, POSITION, MINSIZE, MAXSIZE, WID, TIP, RASTERSIZE, ZORDER, VISIBLE: also accepted.

Callbacks

COMPLETED_CB: action generated when a page successfully completed. Can be called multiple times when a frame set loads its frames, or when a page loads also other pages.

```
int function(Ihandle* ih, char* url);  [in C]
ih:completed_cb(url) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
url: the URL address that completed.

ERROR_CB: action generated when page load fail.

```
int function(Ihandle* ih, char* url);  [in C]
ih:error_cb(url) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
url: the URL address that caused the error.

NAVIGATE_CB: action generated when the browser requests a navigation to another page. It is called before navigation occurs. Can be called multiple times when a frame set loads its frames, or when a page loads also other pages.

```
int function(Ihandle* ih, char* url);  [in C]
ih:navigate_cb(url) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
url: the URL address to navigate to.

Returns: IUP_IGNORE will abort navigation (since 3.4).

NEWWINDOW_CB: action generated when the browser requests a new window.

```
int function(Ihandle* ih, char* url);  [in C]
ih:newwindow_cb(url) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.
url: the URL address that is opened in the new window.

MAP_CB, UNMAP_CB, DESTROY_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB: All common callbacks are supported.

Notes

To learn more about WebKit and WebKitGTK+:

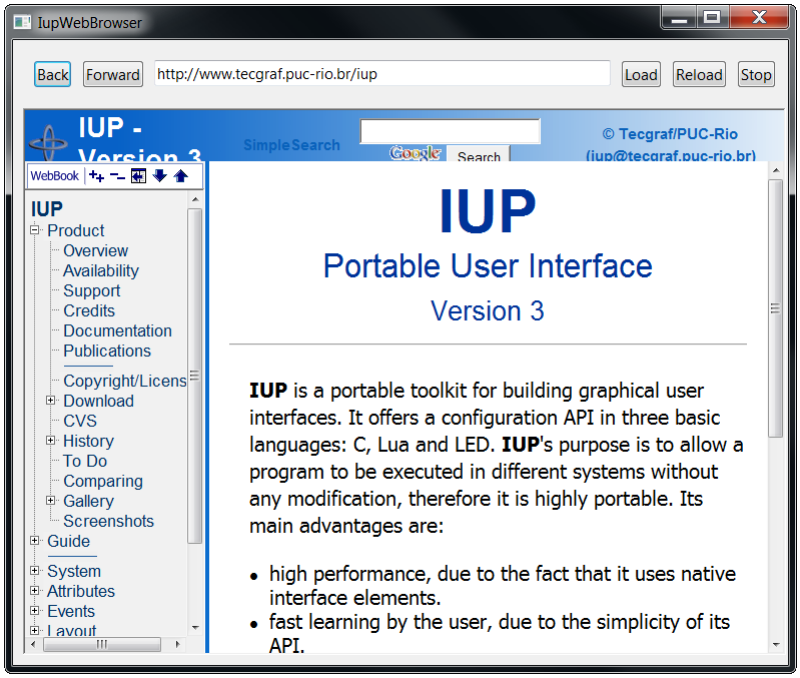
The WebKit Open Source Project
The WebKitGTK+ web page
WebKitGTK+ wiki
WebKitGTK+ tracker

To learn more about Internet Explorer WebBrowser control and ATL:

WebBrowser Control from C/C++
Creating ATL sinks
Examples of sinking COM events

Examples

Browse for Example Files



IupMap

Creates (**maps**) the native interface objects corresponding to the given IUP interface elements.

It will also called recursively to create the native element of all the children in the element's tree.

The element must be already **attached** to a mapped container, except the dialog. A child can only be mapped if its parent is already mapped.

This function is automatically called before the dialog is shown in **IupShow**, **IupShowXY** or **IupPopup**.

Parameters/Return

```
int IupMap(Ihandle* ih); [in C]
iup.Map(ih: ihandle) -> ret: number [in Lua]
or ih:map() [in Lua]
```

ih: Identifier of an interface element.

Returns: IUP_NOERROR if successful. If the element was already mapped returns IUP_NOERROR. If the native creation failed returns IUP_ERROR.

Notes

If the element is a dialog then the abstract layout will be updated even if the dialog is already mapped. If the dialog is visible the elements will be immediately repositioned. Calling **IupMap** for an already mapped dialog is the same as only calling **IupRefresh** for the dialog.

Calling **IupMap** for an already mapped element that is not a dialog does nothing.

If you add new elements to an already mapped dialog you must call **IupMap** for that elements. And then call **IupRefresh** to update the dialog layout.

If the WID attribute of an element is NULL, it means the element was not already mapped. Some containers do not have a native element associated, like VBOX and HBOX. In this case their WID is a fake value (void*)(-1).

It is useful for the application to call **IupMap** when the value of the WID attribute must be known, i.e. the native element must exist, before a dialog is made visible.

The MAP_CB callback is called at the end of the **IupMap** function, after all processing, so it can also be used to create other things that depend on the WID attribute. But notice that for non dialog elements it will be called before the dialog layout has been updated, so the element current size will still be 0x0 (since 3.14).

See Also

IupAppend, IupDetach, IupUnmap, IupCreate, IupDestroy, IupShowXY, IupShow, IupPopup, MAP_CB

IupUnmap (since 3.0)

Unmap the element from the native system. It will also unmap all its children.

It will NOT **detach** the element from its parent, and it will NOT **destroy** the IUP element.

Parameters/Return

```
void IupUnmap(Ihandle* ih); [in C]
iup.Unmap(ih: ihandle) [in Lua]
or ih:unmap() [in Lua]
```

ih: Identifier of an interface element.

Notes

When the element is mapped some attributes are stored only in the native system. If the element is **unmaped** those attributes are lost. Use the function IupSaveClassAttributes when you want to **unmap** the element and keep its attributes.

See Also

IupAppend, IupDetach, IupMap, IupCreate, IupDestroy

IupCreate

Creates an interface element given its class name and parameters. This function is called from all constructors like IupDialog(...), IupLabel(...), and so on.

After **creation** the element still needs to be **attached** to a container and **mapped** to the native system so it can be visible.

Parameters/Return

```
Ihandle* IupCreate(const char *classname); [in C]
Ihandle* IupCreatev(const char *classname, void **params)
Ihandle *IupCreatep(const char *classname, void* params0, ...)
[Not available in Lua]
```

classname: class name of the element to be created
params: list of parameters limited by a NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

See Also

IupAppend, IupDetach, IupMap, IupUnmap, IupDestroy, IupGetClassName

IupDestroy

Destroys an interface element and all its children. Only dialogs, timers, popup menus and images should be normally destroyed, but **detached** controls can also be destroyed.

Parameters/Return

```
void IupDestroy(Ihandle *ih); [in C]
iup.Destroy(ih: ihandle) [in Lua]
or ih:destroy() [in Lua]
```

ih: Identifier of the interface element to be destroyed.

Notes

It will automatically **unmap** and **detach** the element if necessary, and then **destroy** the element.

This function also deletes the main names associated to the interface element being destroyed, but if it has more than one name then some names may be left behind.

Menu bars associated with dialogs are automatically destroyed when the dialog is destroyed.

Images associated with controls are NOT automatically destroyed, because images can be reused in several controls the application must destroy them when they are not used anymore.

All dialogs and all elements that have names are automatically destroyed in **IupClose**.

See Also

IupAppend, IupDetach, IupMap, IupUnmap, IupCreate

IupGetAllClasses (Since 3.3)

Returns the names of all registered classes.

Parameters/Return

```
int IupGetAllClasses(char** names, int max_n); [in C]
iup.GetAllClasses([max_n: number]) -> (names: table, n: number) [in Lua]
```

names: table receiving the names. Only the list of names need to be allocated. Each name will point to an internal string.
max_n: maximum number of names the table can receive. Can be omitted in Lua.

Returns: the actual number of names loaded to the table or -1 (nil) if class not found. If names==NULL or max_n==0 then returns the maximum number of names.

See Also

IupGetClassName, IupGetClassType, IupGetAllAttributes

IupGetClassName (renamed from IupGetType in 2.7)

Returns the name of the class of an interface element.

Parameters/Return

```
char* IupGetClassName(Ihandle* ih); [in C]
iup.GetClassName(ih: ihandle) -> (name: string) [in Lua]
```

ih: Identifier of the interface element.

Returns: the name of the class.

Notes

The following names are known:

```
"image"
"button"
"canvas"
"dialog"
"fill"
"frame"
"hbox"
"item"
```

```
"separator"
"submenu"
"label"
"list"
"menu"
"radio"
"text"
"toggle"
"vbox"
"zbox"
"multiline"
"user"
"matrix"
"tree"
"dial"
"gauge"
"val"
"glcanvas"
"tabs"
"cells"
"colorbrowser"
"colorbar"
"spin"
"sbox"
"cbox"
"progressbar"
"olecontrol"
```

See Also

IupClassMatch, IupGetClassType, IupGetClassAttributes

IupGetClassType (Since 3.0)

Returns the name of the native type of an interface element.

Parameters/Return

```
char* IupGetClassType(Ihandle* ih); [in C]
iup.GetClassType(ih: ihandle) -> (name: string) [in Lua]
```

ih: Identifier of the interface element.

Returns: the class type.

Notes

There are only a few pre-defined class types:

```
"void" - No native representation - HBOX, VBOX, ZBOX, FILL, RADIO
"control" - Native controls - BUTTON, LABEL, TOGGLE, LIST, TEXT, MULTILINE, ITEM, SEPARATOR, SUBMENU, FRAME, others
"canvas" - Drawing canvas, also used as a base control for custom controls.
"dialog"
"image"
"menu"
```

See Also

IupGetClassName, IupGetClassAttributes

IupClassMatch (since 3.4)

Checks if the give class name matches the class name of the given interface element.

Parameters/Return

```
int IupClassMatch(Ihandle* ih, const char* classname); [in C]
iup.ClassMatch(ih: ihandle, classname: string) -> (ret: boolean) [in Lua]
```

ih: Identifier of the interface element.
classname: name of the class to match.

Returns: true (1) if the given name matches the class name or one of its parent class names, false (0) or else.

See Also

IupGetClassName

IupGetClassAttributes (Since 3.0)

Returns the names of all registered attributes of a class.

Parameters/Return

```
int IupGetClassAttributes(const char* classname, char** names, int max_n); [in C]
iup.GetClassAttributes(classname: string[, max_n: number]) -> (names: table, n: number) [in Lua]
```

classname: name of the class
names: table receiving the names. Only the list of names need to be allocated. Each name will point to an internal string.
max_n: maximum number of names the table can receive. Can be omitted in Lua.

Returns: the actual number of names loaded to the table or -1 (nil) if class not found. If names==NULL or max_n==0 then returns the maximum number of names.

See Also

IupGetClassName, IupGetClassType, IupGetAllAttributes

IupGetClassCallbacks (Since 3.3)

Returns the names of all registered callbacks of a class.

Parameters/Return

```
int IupGetClassCallbacks(const char* classname, char** names, int max_n); [in C]
iup.GetClassCallbacks(classname: string[, max_n: number]) -> (names: table, n: number) [in Lua]
```

classname: name of the class
names: table receiving the names. Only the list of names need to be allocated. Each name will point to an internal string.
max_n: maximum number of names the table can receive. Can be omitted in Lua.

Returns: the actual number of names loaded to the table or -1 (nil) if class not found. If names==NULL or max_n==0 then returns the maximum number of names.

See Also

IupGetClassName, IupGetClassType, IupGetAllAttributes

IupSaveClassAttributes

Saves all registered attributes on the internal hash table.

Parameters/Return

```
void IupSaveClassAttributes(Ihandle* ih); [in C]
iup.SaveClassAttributes(ih: ihandle) [in Lua]
```

ih: identifier of the interface element.

Notes

When the element is mapped some attributes are stored only in the native system. If the element is **unmaped** those attributes are lost. So this function is useful when you want to **unmap** the element and keep its attributes.

It will not save id dependent attributes, like those which has a complementary number. For example: items in a **IupList**, **IupTree** or **IupMatrix**.

See Also

IupGetClassAttributes, IupGetClassName, IupGetClassType, IupGetAllAttributes, IupCopyClassAttributes

IupCopyClassAttributes

Copies all registered attributes from one element to another. Both elements must be of the same class.

Parameters/Return

```
void IupCopyClassAttributes(Ihandle* src_ih, Ihandle* dst_ih); [in C]
iup.CopyClassAttributes(src_ih, dst_ih: ihandle) [in Lua]
```

src_ih: identifier of the source element.
dst_ih: identifier of the destiny element.

See Also

IupGetClassAttributes, IupGetClassName, IupGetClassType, IupGetAllAttributes, IupSaveClassAttributes

IupSetClassDefaultAttribute (Since 3.0)

Changes the default value of an attribute for a class. It can be any attribute, i.e. registered attributes or user custom attributes.

Parameters/Return

```
void IupSetClassDefaultAttribute(const char* classname, const char *name, const char *value); [in C]
iup.SetClassDefaultAttribute(classname, name, value: string) [in Lua]
```

classname: name of the class
name: name of the attribute
value: new default value.

Notes

If the value is DEFAULTFONT, DLGBGCOLOR, DLGFGCOLOR, TXTBGCOLOR, TXTFGCOLOR, LINKFGCOLOR or MENUBGCOLOR then the actual default value will be the global attribute of the same name consulted at the time the attribute is consulted.

Attributes that are not strings and attributes that have variable names, like those which has a complementary number, can NOT have a default value. Some attributes can NOT have a default value by definition.

If the new default value is (char*)-1, then the default value is set to be the system default if any is defined.

See Also

IupGetClassName, IupGetClassType, IupGetAllAttributes

IupUpdate
IupUpdateChildren

Mark the element or its children to be redraw when the control returns to the system.

Parameters/Return

```
void IupUpdate(Ihandle* ih); [in C]
void IupUpdateChildren(Ihandle* ih); [in C]
iup.Update(ih: ihandle) [in Lua]
iup.UpdateChildren(ih: ihandle) [in Lua]
```

ih: identifier of the interface element.

See Also

IupRedraw

IupRedraw (since 3.0)

Force the element and its children to be redraw immediately.

Parameters/Return

```
void IupRedraw(Ihandle* ih, int children); [in C]
iup.Redraw(ih: ihandle, children: number) [in Lua]
```

ih: identifier of the interface element.
children: flag to update its children. Can be 0 or 1.

See Also

IupUpdate

IupConvertXYToPos (since 3.0)

Converts a (x,y) coordinate in an item position.

Parameters/Return

```
int IupConvertXYToPos(Ihandle *ih, int x, int y); [in C]
iup.ConvertXYToPos(ih: ihandle, x, y: number) -> (ret: number) [in Lua]
```

ih: Identifier of the element.
x: X coordinate relative to the left corner of the element.
y: Y coordinate relative to the top corner of the element.

Returns: the position starting at 0 (except for **IupList** that starts at 1). If fails returns -1.

Notes

It can be used for **IupText** and **IupScintilla** (returns a position in the string), **IupList** (returns an item), **IupTree** (returns a node identifier) or **IupMatrix** (returns a cell position, where pos=lin*numcol + col).

See Also

IupText, IupList, IupTree, IupMatrix, IupScintilla

Resources

Resources are several auxiliary tools including menus, images, fonts and global names.

Some objects like menus and images, that are not inserted in a dialog children tree, are in fact "associated" with dialogs or controls.

Menus can be associated with dialogs only. Images can be associated with labels, buttons, toggles and menu items (this last in Windows only).

Both images and menus to be associated use a global table of names. This exist because of the LED scripts. First you associate the image or menu Ihandle to a name, then you associated the MENU or IMAGE attribute to the respective name.

For example, in C:

```
Ihandle* img = IupImage (11, 11, pixmap) ;
IupSetHandle("myImg", img);
IupSetAttribute(myButton, "IMAGE", "myImg") ;
```

or in LED:

```
myImg = image[...] (
...
)
myButton = button[IMAGE = myImg]("")
```

or in Lua:

```
myImg = iupimage {
...
}
myButton = iupbutton { title = "", image = myImg }
```

The IupView application is capable of converting several image formats into an **IupImage**, and save an **IupImage** as LED, Lua or ICO.

Only dialogs, timers, popup menus and images can be destroyed. Menu bars associated with dialogs are automatically destroyed.

LED

LED is a dialog-specification language whose purpose is not to be a complete programming language, but rather to make dialog specification simpler than in C.

In LED, attributes and expressions follow this form:

elem = element[**attribute1=value1,attribute2=value2,**...](...expression...)

The names of the elements must not contain the "iup" prefix. Attribute values are always interpreted as strings, but they need to be in quotes ("...") only when they include spaces. The "IUP_"

prefix must not be added to the names of the attributes and predefined values. Expressions contain parameters for creating the element.

In LED there is no distinction between upper and lower case, except for attribute names.

Also there is no NULL definition, so there is no optional parameters, in arrays at least one parameter must exist .

The IupLoad function can parse a text file and create the IUP elements defined in it. Naturally, the same file cannot be loaded more than once, because the elements would be created again. The file parse does not map the elements to the native system.

The LED files are dynamically loaded and must be packaged together with the application's executable. However, this often becomes an inconvenience. To deal with it, there is the LEDC compiler that creates a C module from the LED contents.

To simply view a LED file objects use the LED Viewer application called IupView, in the applications included in the distribution. Pre-compiled binaries are available at the Download.

IupLoad and IupLoadBuffer

Compiles a LED specification.

Parameters/Return

```
char *IupLoad(const char *filename); [in C]
iup.Load(filename: string) -> error: string [in Lua]

char *IupLoadBuffer(const char *buffer); [in C] (since 3.0)
iup.LoadBuffer(buffer: string) -> error: string [in Lua]
```

filename: name of the file containing the LED specification.
buffer: string with the LED specification.

Returns: NULL (nil in Lua) if the file was successfully compiled; otherwise it returns a pointer to a string containing the error message.

Notes

Each time the function loads a LED file, the elements contained in it are created. Therefore, the same LED file cannot be loaded several times, otherwise the elements will also be created several times (the same applies for running Lua files several times).

IupMapFont (Deprecated since 3.0)

Retrieves the name of a native font, given the name of the old IUP FONT names. See the old Font Names table for a list of names.

Parameters/Return

```
char* IupMapFont(const char *iupfont); [in C]
iup.MapFont(iupfont : string) -> (driverfont : string) [in Lua]
```

Returns: the name of the native font.

See Also

IupUnMapFont

IupUnMapFont (Deprecated since 3.0)

Retrieves the name of the old IUP FONT names, given the native font. See the old Font Names table for a list of names.

Parameters/Return

```
char* IupUnMapFont(const char *driverfont); [in C]
iup.UnMapFont(driverfont :string) -> (iupfont : string) [in Lua]
```

Returns: the name of the IUP font, given the native font. If such font does not exist, the function will return NULL.

See Also

IupMapFont

IupImage, IupImageRGB, IupImageRGBA

Creates an image to be shown on a label, button, toggle, or as a cursor.

(**IupImageRGB** and **IupImageRGBA,** since 3.0)

Creation

```
Ihandle* IupImage(int width, int height, const unsigned char *pixels); [in C]
Ihandle* IupImageRGB(int width, int height, const unsigned char *pixels); [in C]
Ihandle* IupImageRGBA(int width, int height, const unsigned char *pixels); [in C]

iup.image{line0: table, line1: table, ...; colors = colors: table} -> (ih: ihandle) [in Lua]
iup.image{width = width: number, height = height: number, pixels = pixels: table, colors = colors: table} -> (ih: ihandle) [in Lua]
iup.imagergb {width = width: number, height = height: number, pixels = pixels: table} -> (ih: ihandle) [in Lua]
iup.imagergba{width = width: number, height = height: number, pixels = pixels: table} -> (ih: ihandle) [in Lua]

image(width, height, pixel0, pixel1, ...) [in LED]
imagergb(width, height, pixel0, pixel1, ...) [in LED]
imagergba(width, height, pixel0, pixel1, ...) [in LED]
```

width: Image width in pixels.
height: Image height in pixels.
pixels: Vector containing the value of each pixel. **IupImage** uses 1 value per pixel, **IupImageRGB** uses 3 values and **IupImageRGBA** uses 4 values per pixel. Each value is always 8 bit. Origin is at the top-left corner and data is oriented top to bottom, and left to right. The pixels array is duplicated internally so you can discard it after the call.
pixel0, **pixel1**, **pixel2**, ...: Value of the pixels. But for **IupImageRGB** and **IupImageRGBA** in fact will be one value for each color channel (pixel_r_0, pixel_g_0, pixel_b_0, pixel_r_1, pixel_g_1, pixel_b_1, pixel_r_2, pixel_g_2, pixel_b_2, ...).
line0, **line1**: unnamed tables, one for each line containing pixels values. See Notes below.

colors: table named colors containing the colors indices. See Notes below.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

"0" Color in index 0.
"1" Color in index 1.
"2" Color in index 2.
...
"i" Color in index i.

The indices can range from 0 to 255. The total number of colors is limited to 256 colors. Notice that in Lua the first index in the array is "1", the index "0" is ignored in IupLua. Be careful when setting colors, since they are attributes they follow the same storage rules for standard attributes.

The values are integer numbers from 0 to 255, one for each color in the RGB triple (For ex: "64 190 255"). If the value of a given index is "BGCOLOR", the color used will be the background color of the element on which the image will be inserted. The "BGCOLOR" value must be defined within an index less than 16.

Used only for images created with **IupImage**.

AUTOSCALE: automatically scale the image by a given real factor. If not defined the global attribute IMAGEAUTOSCALE will be used. Values are the same of the global attribute. (since 3.16)

BGCOLOR: The color used for transparency. If not defined uses the BGCOLOR of the control that contains the image.

BPP (read-only): returns the number of bits per pixel in the image. Images created with **IupImage** returns 8, with **IupImageRGB** returns 24 and with **IupImageRGBA** returns 32. (since 3.0)

CHANNELS (read-only): returns the number of channels in the image. Images created with **IupImage** returns 1, with **IupImageRGB** returns 3 and with **IupImageRGBA** returns 4. (since 3.0)

HEIGHT (read-only): Image height in pixels.

HOTSPOT: Hotspot is the position inside a cursor image indicating the mouse-click spot. Its value is given by the x and y coordinates inside a cursor image. Its value has the format "x:y", where x and y are integers defining the coordinates in pixels. Default: "0:0".

RASTERSIZE (read-only): returns the image size in pixels. (since 3.0)

WID (read-only): returns the internal pixels data pointer. (since 3.0)

WIDTH (read-only): Image width in pixels.

Notes

Application icons are usually 32x32. Toolbar bitmaps are 24x24 or smaller. Menu bitmaps and small icons are 16x16 or smaller.

Images created with the **IupImage*** constructors can be reused in different elements.

The images should be destroyed when they are no longer necessary, by means of the **IupDestroy** function. To destroy an image, it cannot be in use, i.e the controls where it is used should be destroyed first. Images that were associated with controls by names are automatically destroyed in IupClose.

Please observe the rules for creating cursor images: CURSOR.

Usage

Images are used in elements such as buttons and labels by attributes that points to names registered with IupSetHandle. You can also use **IupSetAttributeHandle** to shortcut the set of an image as an attribute. For example:

```
Ihandle* image = IupImage(width, height, pixels);

IupSetHandle("MY_IMAGE_NAME", image);
IupSetAttribute(label, "IMAGE", "MY_IMAGE_NAME");
or
IupSetAttributeHandle(label, "IMAGE", image); // an automatic name will be created internally
```

In Windows, names of resources in RC files linked with the application are also accepted. In GTK, names of GTK Stock Items are also accepted. In Motif, names of bitmaps installed on the system are also accepted. For example:

```
IupSetAttribute(label, "IMAGE", "TECGRAF_BITMAP");  // available in the "etc/iup.rc" file
or
IupSetAttribute(label, "IMAGE", "gtk-open");  // available in the GTK Stock Items
```

In all drivers, a path to a file name can also be used as the attribute value (since 3.0). But the available file formats supported are system dependent. The Windows driver supports BMP, ICO and CUR. The GTK driver supports the formats supported by the GDK-PixBuf library, such as BMP, GIF, JPEG, PCX, PNG, TIFF and many others. The Motif driver supports the X-Windows bitmap. For example:

```
IupSetAttribute(label, "IMAGE", "../etc/tecgraf.bmp");
```

A more format independent approach can be reached using the IUP-IM Functions.

Colors

In Motif, the alpha channel in RGBA images is always composed with the control BGCOLOR by IUP prior to setting the image at the control. In Windows and in GTK, the alpha channel is composed internally by the system. But in Windows for some controls the alpha must be composed a priori also, it includes: **IupItem** and **IupSubmenu** always; and **IupToggle** when NOT using Visual Styles. This implies that if the control background is not uniform then probably there will be a visible difference where it should be transparent.

For **IupImage**, if a color is not set, then it is used a default color for the 16 first colors. The default color table is the same for Windows, GTK and Motif:

```
 0 =   0,  0,  0 (black)
 1 = 128,  0,  0 (dark red)
 2 =   0,128,  0 (dark green)
 3 = 128,128,  0 (dark yellow)
 4 =   0,  0,128 (dark blue)
 5 = 128,  0,128 (dark magenta)
 6 =   0,128,128 (dark cian)
 7 = 192,192,192 (gray)
 8 = 128,128,128 (dark gray)
 9 = 255,  0,  0 (red)
10 =   0,255,  0 (green)
11 = 255,255,  0 (yellow)
12 =   0,  0,255 (blue)
13 = 255,  0,255 (magenta)
14 =   0,255,255 (cian)
15 = 255,255,255 (white)
```

For images with more than 16 colors, and up to 256 colors, all the color indices must be defined up to the maximum number of colors. For example, if the biggest image index is 100, then all the

colors from i=16 up to i=100 must be defined even if some indices are not used.

Samples

You can obtain several images from the IupImageLib, a library of pre-defined images. To view the images you can use the IupView in the applications included in the distribution, available at the Download. IupView is also capable of converting several image formats into an **IupImage**, and save IUP images as LED, Lua or ICO.

The EdPatt and the IMLAB applications can load and save images in simplified LED format. They allow operations such as importing GIF images and exporting them as IUP images. **EdPatt** allows you to manually edit the images, and also have support for images in IupLua.

IupLua Old Constructor

In Lua, the 8bpp image can also be created using an unnamed table, using a series of tables for each line. Width and height will be guessed from the tables sizes. For example:

```
img = iup.image{
  { 1,2,3,3,3,3,3,3,2,1 },
  { 2,1,2,3,3,3,3,3,2,1,2 },
  { 3,2,1,2,3,3,3,2,1,2,3 },
  { 3,3,2,1,2,3,2,1,2,3,3 },
  { 3,3,3,2,1,2,1,2,3,3,3 },
  { 3,3,3,3,2,1,2,3,3,3,3 },
  { 3,3,3,2,1,2,1,2,3,3,3 },
  { 3,3,2,1,2,3,2,1,2,3,3 },
  { 3,2,1,2,3,3,3,2,1,2,3 },
  { 2,1,2,3,3,3,3,3,2,1,2 },
  { 1,2,3,3,3,3,3,3,3,2,1 };
  colors = {
    "0 1 0",      -- index 1
    "255 0 0",    -- index 2
    "255 255 0"   -- index 3
  }
}
```

Using this constructor the image data can NOT has 0 indices. Notice that the indexing of the unamed tables is different than the **colors** table. The first value in the **colors** table is relative to the color index 1, but the first value of the unamed tables is relative to the pixel 0.

After the image is created in Lua, the unnamed tables are not accessible anymore, since "img[1]" will return the attribute "1" value which is the color "0 1 0". To access the original table values you must use "raawget" function, for example:

```
lin0 = rawget(img, 1)  -- line index 0
lin1 = rawget(img, 2)  -- line index 1
lin2 = rawget(img, 3)  -- line index 2
...
pixel0 = lin0[1]       -- column index 0
pixel1 = lin0[2]       -- column index 1
pixel3 = lin0[3]       -- column index 3
...
```

IupLua New Constructors

The new constructors since IUP 3 are different and must contains explicit values for **width**, **height** and **pixels**. Also the indexing of the **colors** table is the same of the **pixels** table, the first value is the color index 0. For example:

```
img = iup.image{
  width = 11,
  height = 11,
  pixels = {
   1,2,0,0,0,0,0,0,0,2,1,
   2,1,2,0,0,0,0,0,2,1,2,
   0,2,1,2,0,0,0,2,1,2,0,
   0,0,2,1,2,0,2,1,2,0,0,
   0,0,0,2,1,2,1,2,0,0,0,
   0,0,0,0,2,1,2,0,0,0,0,
   0,0,0,2,1,2,1,2,0,0,0,
   0,0,2,1,2,0,2,1,2,0,0,
   0,2,1,2,0,0,0,2,1,2,0,
   2,1,2,0,0,0,0,0,2,1,2,
   1,2,0,0,0,0,0,0,0,2,1},
  colors = {
    "255 255 0"   -- index 0
    "0 1 0",      -- index 1
    "255 0 0",    -- index 2
  }
}
```

Although in Lua they are still referenced as index 1, so img.**colors**[1] returns the color of the index 0 in the image.

Here is the same image but using 24bpp:

```
img = iup.imagergb{
  width = 11,
  height = 11,
  pixels = {
    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,
  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,
  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,
  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,
  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,
  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,
  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,
  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,
  255,  0,0,    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0,  255,  0,0,
    0,255,0,  255,  0,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,255,0,  255,  0,0,    0,255,0}
}
```

Then at the **pixels** table we have:

```
r0 = img.pixels[1]   g0 = img.pixels[2]   b0 = img.pixels[3]
r1 = img.pixels[4]   g1 = img.pixels[5]   b1 = img.pixels[6]
r3 = img.pixels[7]   g3 = img.pixels[8]   b3 = img.pixels[9]
...
```

If the image was created in C then there is no way to access its pixels values in Lua, except as an userdata using the WID attribute.

Examples

Browse for Example Files

See Also

IupLabel, IupButton, IupToggle, IupDestroy.

IupImageLib

A library of pre-defined stock images for buttons and labels.

Initialization

To generate an application that uses this function, the program must be linked to the functions library (**iupimglib**.lib on Windows and lib**iupimglib**.a on Unix).

Reference

```
void IupImageLibOpen(void); [in C]
iup.ImageLibOpen()  [in Lua]
```

This function register the names but do not load the images. The images will be loaded only if they are used in a control. The loaded images will be automatically released at IupClose.

In Lua, when require"iupluaimglib" is used this function will be automatically called.

Usage

The following names can be used after the library initialization. The names are NOT registered using **IupSetHandle**, they will be automatically loaded when associated with a control.

Base Library Group

These bitmaps are 16x16-8bpp (Motif) and 32x32-32bpp (Windows) pixels size images that can be used in Buttons, usually inside toolbars. GTK has several image sizes available: 16x16, 18x18, 20x20, 24x24, 32x32, and 48x48. GTK images displayed here are just a sample with the 24x24-32bpp pixels size, they actually depends on the current GTK theme used by the system.

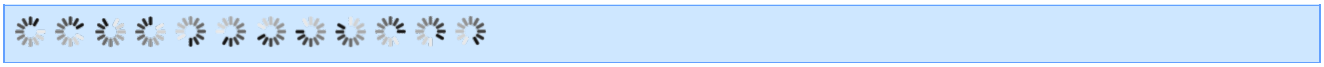
In Windows and GTK, to force a specific height for the images use the IMAGESTOCKSIZE global attribute (since 3.16), it can have the following values 16, 24, 32 and 48. In GTK if the system does not provides the image in that size it will be automatically resized from an existing size. In Windows if the image is not 32x32 then it will be automatically resized. The default size depends on the screen resolution: 96 DPI = 16, 144 DPI = 24, 192 DPI = 32, 288 DPI = 48.

Name	Image (Motif)	Image (GTK)	Image (Windows)
"IUP_ActionCancel"			
"IUP_ActionOk"			
"IUP_ArrowDown"			
"IUP_ArrowLeft"			
"IUP_ArrowRight"			
"IUP_ArrowUp"			
"IUP_EditCopy"			
"IUP_EditCut"			
"IUP_EditErase"			
"IUP_EditFind"			
"IUP_EditPaste"			
"IUP_EditRedo"			
"IUP_EditUndo"			
"IUP_FileClose"			
"IUP_FileNew"			
"IUP_FileOpen"			
"IUP_FileProperties"			
"IUP_FileSave"			
"IUP_MediaForward"			
"IUP_MediaGoToBegin"			
"IUP_MediaGoToEnd"			
"IUP_MediaPause"			
"IUP_MediaPlay"			
"IUP_MediaRecord"			
"IUP_MediaReverse"			
"IUP_MediaRewind"			

"IUP_MediaStop"			
"IUP_MessageError"			
"IUP_MessageHelp"			
"IUP_MessageInfo"			
"IUP_NavigateHome"			
"IUP_NavigateRefresh"			
"IUP_Print"			
"IUP_PrintPreview"			
"IUP_ToolsColor"			
"IUP_ToolsSettings"			
"IUP_ToolsSortAscend"			
"IUP_ToolsSortDescend"			
"IUP_ViewFullScreen"			
"IUP_ZoomActualSize"	1:1		
"IUP_ZoomIn"			
"IUP_ZoomOut"			
"IUP_ZoomSelection"			

The following images were removed from the pre-compiled library (since 3.16): "IUP_FileCloseAll", "IUP_FileSaveAll", "IUP_FileText", "IUP_FontBold", "IUP_FontDialog", "IUP_FontItalic", "IUP_WindowsCascade", "IUP_WindowsTile", "IUP_Zoom". Although their C source code is still available.

"IUP_CircleProgressAnimation" animation added in version 3.17. To be used in **IupAnimatedLabel** to show indefinite progress. It has 12 frames of 32x32 with 32bpp, and FRAMETIME is set to 83ms (approximately 1 second for a full turn).



Logo Group 32x32

These images are 32x32-32bpp pixels size (or just 32 pixels height) images that can be used in Labels, usually inside toolbars.

Name	Image (Generic)
"IUP_Tecgraf"	
"IUP_PUC-Rio"	
"IUP_BR"	
"IUP_Lua"	

Name	Image (Generic)
"IUP_TecgrafPUC-Rio"	
"IUP_Petrobras"	

Logo Group 48x48 (NOT included in the pre-compiled library, since 3.3)

These images are 48x48-32bpp pixels size (or just 48 pixels height) images that can be used in Labels, usually inside dialogs.

Name	Image (Generic)
"IUP_LogoTecgraf"	
"IUP_LogoPUC-Rio"	
"IUP_LogoBR"	
"IUP_LogoLua"	

Name	Image (Generic)
"IUP_LogoTecgrafPUC-Rio"	
"IUP_LogoPetrobras"	

Icon Group 48x48 (NOT included in the pre-compiled library, since 3.3)

Here are other images available in the IUP stock library, commonly used by the respective systems. All images are 48x48-32bpp pixels size.

Name	Image (Windows)
"IUP_DeviceCamera"	

Name	Image (Windows)	Image (GTK)
"IUP_IconMessageSecurity"		
"IUP_IconMessageWarning"		
"IUP_IconMessageInfo"		
"IUP_IconMessageError"		
"IUP_IconMessageHelp"		

"IUP_DeviceCD"	
"IUP_DeviceCellPhone"	
"IUP_DeviceComputer"	
"IUP_DeviceFax"	
"IUP_DeviceHardDrive"	
"IUP_DeviceMP3"	
"IUP_DeviceNetwork"	
"IUP_DevicePDA"	
"IUP_DevicePrinter"	
"IUP_DeviceScanner"	
"IUP_DeviceSound"	
"IUP_DeviceVideo"	

Icon Group 32x32 (NOT available as C source code) (since 3.16)

Here are other images available in the IUP stock library, commonly used in Windows. All images are 32x32-32bpp pixels size.

Image (Windows)

Notes

All images are available as PNG files in the "iup/html/en/imglib/*" folder. Except for the GTK images that are only sample images.

Use the IupView application to export PNG images to C, LED or Lua.

All 8bpp images are from the old ImageLib and since Motif does not have any stock images, we selected this set to be used in Motif. Although the IUP Motif driver supports 32bpp images.

The pre-compiled library does not include images larger than 48x48 (inclusive). If you want to use them you must include their source code, or re-compile the library defining USE_IUP_IMGLIB_LARGE during compilation, for example: "make USE_IUP_IMGLIB_LARGE=1".

Not available in AIX.

All "Windows" images copyright Microsoft and were extracted from the Visual Studio 2013 Image Library. Their use **must** be used consistently with their description in the Visual Studio Image Library, and so consistently with the IUP name. These files are available for free on the link: Visual Studio Image Library. You can find a copy of the license terms here: "Visual Studio 2013 Image Library EULA.docx". The most important statement on this document is: "Media Elements. You may copy and use images in the Image Library provided with the software and identified for such use in documents and projects that you create."

GTK stock images are released under the GTK license.

PUC-Rio, Tecgraf/PUC-Rio, Petrobras and Lua images are copyright of the respective companies or owners. The Petrobras logo images follow the company established rules.

Lua image graphic design by A. Nakonechnyj. Copyright © 1998. All rights reserved.

See Also

IupImage

IUP-IM Functions

Functions to load/save an **IupImage** from/to a file using the IM library. The function can load or save the formats: BMP, JPEG, GIF, TIFF, PNG, PNM, PCX, ICO and others. For more information about the IM library see http://www.tecgraf.puc-rio.br/im.

Initialization and Usage

To generate an application that uses this function, the program must be linked with the IM library and with the function library (im and iupim libraries). The "iupim.h" file must also be included in the source code.

To make the functions available in Lua use require"iupluaim" or manually call the initialization function in C, iupimlua_open, after calling **iuplua_open**. When manually calling the function the iupluaim.h file must also be included in the source code and the program must be linked to the iupluaim library.

Load

```
Ihandle* IupLoadImage(const char* file_name); [in C]
iup.LoadImage(file_name: string) -> (elem: ihandle) [in Lua]
```

file_name: Name of the file to be loaded.

Returns: the identifier of the created image, or NULL if an error occurs. When failed the global attribute "IUPIM_LASTERROR" is set with a message describing the error.

Save

```
int IupSaveImage(Ihandle* ih, const char* file_name, const char* format); [in C]
iup.SaveImage(ih: ihandle, file_name, format: string) -> (ret: boolean) [in Lua]
```

ih: handle of the **IupImage**.
file_name: Name of the file to be loaded.
format: format descriptor for IM. For ex: "BMP", "JPEG", "GIF", "TIFF", "PNG", "PNM", "PCX", "ICO", etc.

Returns: zero if failed. When failed the global attribute "IUPIM_LASTERROR" is set with a message describing the error.

LoadAnimation (since 3.17)

```
Ihandle* IupLoadAnimation(const char* file_name); [in C]
iup.LoadAnimation(file_name: string) -> (elem: ihandle) [in Lua]
```

file_name: Name of the file to be loaded.

Returns: the identifier of the created animation, or NULL if an error occurs. When failed the global attribute "IUPIM_LASTERROR" is set with a message describing the error.

An animation is simply an **IupUser** element with several **IupImage** elements as children. The total number of images can be obtained using **IupGetChildCount**. The time between frames is defined by the FRAMETIME attribute if FPS is present on the file.

IM supports loading of multiple images from the same file for the following formats: GIF, TIFF, AVI (additional library) and WMV (additional library).

LoadAnimationFrames (since 3.17)

```
Ihandle* IupLoadAnimationFrames(const char** file_name_list, int file_count); [in C]
iup.LoadAnimationFrames(file_name_list: table of strings, file_count: number) -> (elem: ihandle) [in Lua]
```

file_name_list: List of file names to be loaded.
file_count: number of file names in the list.

Returns: the identifier of the created animation, or NULL if an error occurs. When failed the global attribute "IUPIM_LASTERROR" is set with a message describing the error. The FRAMETIME attribute is not set.

Native Handle to imImage

```
imImage* IupGetNativeHandleImage(void* handle); [in C]
iup.GetNativeHandleImage(handle: userdata) -> (image: imImage) [in Lua]
```

handle: image native handle. In Win32 is a **HANDLE** of a DIB. In GTK is a **GdkPixbuf***. In Motif is a **Pixmap**. Its memory is released after the **imImage** is created. In Lua is a light user data.

Returns: the **imImage*** handle. Useful when pasting data from a **IupClipboard**.

You must include the "im_image.h" header before the "iupim.h" to enable these functions.

imImage to Native Handle

```
imImage* IupGetImageNativeHandle(imImage* image); [in C]
iup.GetImageNativeHandle(image: imImage) -> (handle: userdata) [in Lua]
```

image: the **imImage*** handle. Must be a bitmap image.

Returns: the image native handle. In Win32 is a **HANDLE** for a DIB. In GTK is a **GdkPixbuf***. In Motif is a **Pixmap**. Usefull when copying data to a **IupClipboard**. In Lua is a light user data.

You must include the "im_image.h" header before the "iupim.h" to enable these functions.

imImage to IupImage (since 3.10)

```
Ihandle* IupImageFromImImage(imImage* image); [in C]
iup.ImageFromImImage(image: imImage) -> (elem: ihandle) [in Lua]
```

image: the **imImage*** handle. Must be a bitmap image.

Returns: the IupImage handle.

You must include the "im_image.h" header before the "iupim.h" to enable these functions.

See Also

IupImage, IupSaveImageAsText, IupClipboard

IupSaveImageAsText (since 3.0)

Saves the **IupImage** as a text file to be reused in other programs.

It does NOT depends on the IM library.

Parameters/Return

```
int IupSaveImageAsText(Ihandle* ih, const char* file_name, const char* format, const char* name); [in C]
iup.SaveImageAsText(ih: ihandle, file_name, format[, name]: string) -> (ret: boolean) [in Lua]
```

ih: handle of the **IupImage**.
file_name: Name of the file to be loaded.
format: text format. Can be: "LED", "LUA" or "C".
name: name of the image. Can be NULL.

Returns: zero if failed, non zero value if success.

Notes

If name is NULL and the **IupImage** is associated with a name then that name is used, if no name is associated then "image" will be used.

See Also

IupImage, IUP-IM Functions

Keyboard

The application can control the focus using the functions **IupGetFocus** and **IupSetFocus**. When the focus is changed the application is notified trough the callbacks GETFOCUS_CB and KILLFOCUS_CB.

Keyboard navigation in the dialog uses the "Tab" and "Shilf+Tab" keys to change the keyboard focus from one control to another. The exception is when the focus is at an **IupMultiline** control, to change focus the combination "Ctrl+Tab" must be used, because "Tab" is a valid input for the text. All IUP interactive controls have Tab stops, but the navigation order is related to the order the controls are placed in the dialog and can not be changed. The order is the same implemented by the functions **IupNextField** and **IupPreviousField**. To remove the Tab stop from a control use the CANFOCUS attribute.

Arrows can also be used for navigation between buttons and toggles. This is necessary because when an **IupToggle** is inside an **IupRadio** the "Tab" keys will navigate only to the selected toggle.

In Windows, the focus feedback only appears after the user presses a key (except for the **IupText** where the feedback is the caret). Before pressing a key if you click in a control the focus feedback will be NOT be shown although it will be in focus. **IupMatrix** and other additional controls will always show their focus feedback. In GTK and Motif the focus feedback is always shown for the control that has the focus.

Two keys are also important in keyboard navigation: "Enter" and "Esc". But they are only effective if the application register the attributes DEFAULTENTER and DEFAULTESC of the IupDialog. These attributes configure buttons to be activated when the respective key is pressed. Again "Enter" is a valid key for the Multiline so the combination "Ctrl+Enter" must be used instead. If the focus is at a button then the Enter key will activate that button independent from the DEFAULTENTER attribute.

Usualy the application will process keyboard input in the **IupCanvas** using the KEYPRESS_CB callback. But there is also the K_ANY callback that can be used for all the controls, but it does not have control of the press state, it is called only when the key is pressed. Both callbacks use the key codification explained in Keyboard Codes. These codes are also used in the ACTION callbacks of **IupText** and **IupMultiline**, and in shortcuts using the KEY attribute of **IupItem** and **IupSubmenu**. Finally all the keyboard codes can be used as callback names to implement application hot keys.

Keyboard Codes

The table below shows the IUP codification of common keys in a keyboard. Each key is represented by an integer value, defined in the "iupkey.h" file header, which should be included in the application to use the key definitions. These keys are used in K_ANY and KEYPRESS_CB callbacks to inform the key that was pressed in the keyboard.

From the definition in the table, change the prefix to K_s*, K_c*, K_m* and K_y* to add the respective modifier (Shift, Control, Alt and Sys). Sys in Windows is the Windows key and in Mac is the Apple key. Check the "iupkey.h" file header for all the definitions.

IUP provides definitions only for common control keys and ASCii characters. Other key combinations are accessed using the macros described bellow. Also the global attribute "MODKEYSTATE" can be used to detect the combination of two or more modifiers. Notice that some key combinations are never available because they are restricted by the system. Notice that all of this does not affect the **IupText** and **IupMultiline** text input.

The **iup_isprint(key)** macro informs if a key can be directly used as a printable character. The **iup_isXkey(key)** macro informs if a given key is an extended code. The **iup_isShiftXkey(key)** macro informs if a given key is an extended code using the Shift modifier, the **iup_isCtrlXkey(key)** macro for the Ctrl modifier, the **iup_isAltXkey(key)** macro for the Alt modifier, and the **iup_isSysXkey(key)** macro for the Sys modifier. To obtain a key code for a generic combination you can start with the base key from the table and combine it repeated times using the macros **iup_XkeyShift(key)**, **iup_XkeyCtrl(key)**, **iup_XkeyAlt(key)** and **iup_XkeySys(key)**.

These macros are also available in Lua as a function with the same name (iup.isprint(key), iup.isXkey(key), and so on) and returning a boolean.

Note: GTK in Windows does not generates the Win modifier key, the K_Print and the K_Pause keys (up to GTK version 2.8.18).

Key	Code / Callback
Space	K_SP
!	K_exclam
"	K_quotedbl
#	K_numbersign
$	K_dollar
%	K_percent
&	K_ampersand
'	K_apostrophe
(	K_parentleft
)	K_parentright
*	K_asterisk
+	K_plus
,	K_comma
-	K_minus
.	K_period
/	K_slash
0	K_0
1	K_1
2	K_2
3	K_3
4	K_4

5	K_5
6	K_6
7	K_7
8	K_8
9	K_9
:	K_colon
;	K_semicolon
<	K_less
=	K_equal
>	K_greater
?	K_question
@	K_at
A	K_A
B	K_B
C	K_C
D	K_D
E	K_E
F	K_F
G	K_G
H	K_H
I	K_I
J	K_J
K	K_K
L	K_L
M	K_M
N	K_N
O	K_O
P	K_P
Q	K_Q
R	K_R
S	K_S
T	K_T
U	K_U
V	K_V
W	K_W
X	K_X
Y	K_Y
Z	K_Z
[	K_bracketleft
\	K_backslash
]	K_bracketright
^	K_circum
_	K_underscore
`	K_grave
a	K_a
b	K_b
c	K_c
d	K_d
e	K_e
f	K_f
g	K_g
h	K_h
i	K_i
j	K_j
k	K_k
l	K_l
m	K_m
n	K_n
o	K_o
p	K_p
q	K_q
r	K_r
s	K_s
t	K_t
u	K_u
v	K_v
w	K_w

x	K_x
y	K_y
z	K_z
{	K_braceleft
|	K_bar
}	K_braceright
~	K_tilde
Esc	K_ESC
Enter	K_CR
BackSpace	K_BS
Insert	K_INS
Del	K_DEL
Tab	K_TAB
Home	K_HOME
Up Arrow	K_UP
PgUp	K_PGUP
Left Arrow	K_LEFT
Middle	K_MIDDLE
Right Arrow	K_RIGHT
End	K_END
Down Arrow	K_DOWN
PgDn	K_PGDN
Pause	K_PAUSE
Print Screen	K_Print
Context Menu	K_Menu
´	K_acute
ç	K_ccedilla
¨	K_diaeresis
F1	K_F1
F2	K_F2
F3	K_F3
F4	K_F4
F5	K_F5
F6	K_F6
F7	K_F7
F8	K_F8
F9	K_F9
F10	K_F10
F11	K_F11
F12	K_F12
Left Shift	K_LSHIFT
Right Shift	K_RSHIFT
Left Ctrl	K_LCTRL
Right Ctrl	K_RCTRL
Left Alt	K_LALT
Right Alt	K_RALT
Scroll Lock	K_SCROLL
Num Lock	K_NUM
Caps Lock	K_CAPS

IupNextField

Shifts the focus to the next element that can have the focus. It is relative to the given element and does not depend on the element currently with the focus.

It will search for the next element first in the children, then in the brothers, then in the uncles and their children, and so on.

This sequence is not the same sequence used by the Tab key, which is dependent on the native system.

Parameters/Return

```
Ihandle* IupNextField(Ihandle* ih); [in C]
iup.NextField(ih: ihandle) -> (next: ihandle) [in Lua]
```

ih: identifier of the interface element.

Returns: the element that received the focus or NULL if not found.

See Also

IupPreviousField.

IupPreviousField

Shifts the focus to the previous element that can have the focus. It is relative to the given element and does not depend on the element currently with the focus.

Parameters/Return

```
Ihandle* IupPreviousField(Ihandle* ih); [in C]
iup.PreviousField(ih: ihandle) -> (previous: ihandle) [in Lua]
```

ih: identifier of the interface element.

Returns: the element that received the focus or NULL if not found.

See Also

IupNextField.

IupGetFocus

Returns the identifier of the interface element that has the keyboard focus, i.e. the element that will receive keyboard events.

Parameters/Return

```
Ihandle* IupGetFocus(void); [in C]
iup.GetFocus() -> ih: ihandle [in Lua]
```

Returns: the element with focus or NULL if no element has the focus.

See Also

IupSetFocus

IupSetFocus

Sets the interface element that will receive the keyboard focus, i.e., the element that will receive keyboard events. But this will be processed only after the con

Parameters/Return

```
Ihandle *IupSetFocus(Ihandle *ih); [in C]
iup.SetFocus(ih: ihandle) -> ih: ihandle [in Lua]
```

ih: identifier of the interface element that will receive the keyboard focus. Only elements that can have the keyboard focus, are mapped, active and visible can be used, other elements are ignored.

Returns: the identifier of the interface element that previously had the keyboard focus.

Notes

The value returned by **IupGetFocus** will be updated only after the main loop regain the control and the control actually receive the focus. So if you call **IupGetFocus** right after **IupSetFocus** the return value will be different. You could call **IupFlush** between the two functions to obtain the same value.

See Also

IupGetFocus.

IupItem

Creates an item of the menu interface element. When selected, it generates an action.

Creation

```
Ihandle* IupItem(const char *title, const char *action); [in C]
iup.item{[title = title: string]} -> ih: ihandle [in Lua]
item(title, action) [in LED]
```

title: Text to be shown on the item. It can be NULL. It will set the TITLE attribute.
action: Name of the action generated when the item is selected. It can be NULL.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

AUTOTOGGLE (non inheritable): enables the automatic toggle of VALUE state when the item is activated. Default: NO. (since 3.0)

KEY (non inheritable): Underlines a key character in the submenu title. It is updated only when TITLE is updated. **Deprecated (since 3.0)**, use the mnemonic support directly in the TITLE attribute.

HIDEMARK [Motif and GTK Only]: If enabled the item cannot be checked, since the check box will not be shown. If all items in a menu enable it, then no empty space will be shown in front of the items. Normally the unmarked check box will not be shown, but since GTK 2.14 the unmarked check box is always shown. If your item will not be marked you must set HIDEMARK=YES, since this is the most common case we changed the default value to YES for this version of GTK, but if VALUE is defined the default goes back to NO. Default: NO. (since 3.0)

IMAGE [Windows and GTK Only] (non inheritable): Image name of the check mark image when VALUE=OFF. In Windows, an item in a menu bar cannot have a check mark. Ignored if item in a menu bar. A recommended size would be 16x16 to fit the image in the menu item. In Windows, if larger than the check mark area it will be cropped.

IMPRESS [Windows and GTK Only] (non inheritable): Image name of the check mark image when VALUE=ON.

TITLE (non inheritable): Item text. The "&" character can be used to define a mnemonic, the next character will be used as key. Use "&&" to show the "&" character instead on defining a mnemonic. When in a menu bar an item that has a mnemonic can be activated from any control in the dialog using the "Alt+key" combination.

> The text also accepts the control character '\t' to force text alignment to the right after this character. This is used to add shortcut keys to the menu, aligned to the right, ex: "Save\tCtrl+S", but notice that the shortcut key (also known as Accelerator or Hot Key) still has to be implemented. To implement a shortcut use the K_* callbacks in the dialog.

TITLEIMAGE (non inheritable): Image name of the title image. In Windows, it appears before of the title text and after the check mark area (so both title and title image can be visible). In Motif, it must be at least defined during map, it replaces the text, and only images will be possible to set (TITLE will be hidden). In GTK, it will appear on the check mark area. (since 3.0)

VALUE (non inheritable): Indicates the item's state. When the value is ON, a mark will be displayed to the left of the item. Default: OFF. An item in a menu bar cannot have a check mark. When

IMAGE is used, the checkmark is not shown. See the item AUTOTOGGLE attribute and the menu RADIO attribute.

WID (non inheritable): In Windows, returns the HMENU of the parent menu.

ACTIVE: also accepted.

Callbacks

ACTION: Action generated when the item is selected. IUP_CLOSE will be processed. Even if inside a popup menu when IUP_CLOSE is returned, the current popup dialog or the main loop will be closed.

HIGHLIGHT_CB: Action generated when the item is highlighted.

MAP_CB, UNMAP_CB, DESTROY_CB, HELP_CB: common callbacks are supported.

Notes

Menu items are activated using the Enter key.

In Motif and GTK, the text font will be affected by the dialog font when the menu is mapped.

Since GTK 2.14 to have a menu item that can be marked you must set the VALUE attribute to ON or OFF, or set HIDEMARK=NO, before mapping the control.

Examples

Browse for Example Files

See the **IupMenu** element for screenshots.

See Also

IupSeparator, IupSubmenu, IupMenu.

IupMenu

Creates a menu element, which groups 3 types of interface elements: item, submenu and separator. Any other interface element defined inside a menu will be an error.

Creation

```
Ihandle* IupMenu(Ihandle *child, ...); [in C]
Ihandle* IupMenuv(Ihandle **children); [in C]
iup.menu{child, ...: ihandle} -> (ih: ihandle) [in Lua]
menu(child, ...) [in LED]
```

child, ... : List of identifiers that will be grouped by the menu. NULL must be used to mark the end of the list in C. It can be empty in C or Lua, not in LED.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

BGCOLOR: the background color of the menu, affects all items in the menu. (since 3.0)

RADIO (non inheritable): enables the automatic toggle of one child item. When a child item is selected the other item is automatically deselected. The menu acts like a **IupRadio** for its children. Submenus and their children are not affected.

WID (non inheritable): In Windows, returns the HMENU of the menu.

Callbacks

OPEN_CB: Called just before the menu is opened.

MENUCLOSE_CB: Called just after the menu is closed.

MAP_CB, UNMAP_CB, DESTROY_CB : common callbacks are supported.

Notes

A menu can be a menu bar of a dialog, defined by the dialog's MENU attribute, or a popup menu.

A popup menu is displayed for the user using the **IupPopup** function (usually on the mouse position) and disappears when an item is selected.

IupDestroy should be called only for popup menus. Menu bars associated with dialogs are automatically destroyed when the dialog is destroyed. But if you change the menu of a dialog for another menu, the previous one should be destroyed using **IupDestroy**. If you replace a menu bar of a dialog, the previous menu is unmapped.

Any item inside a menu bar can retrieve attributes from the dialog using **IupGetAttribute**. It is not necessary to call **IupGetDialog**.

The menu can be created with no elements and be dynamic filled using IupAppend or IupInsert.

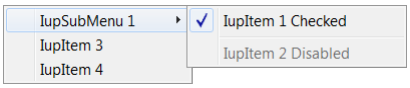
Lua Binding

Offers a "cleaner" syntax than LED for defining menu, submenu and separator items. The list of elements in the menu is described as a string, with one element after the other, separated by commas.

Each element can be:

```
{"<item_name>"} - menu item
{"<submenu_name>", <menu>} - submenu
{} - separator
```

For example:



```
mnu = iup.menu
{
  iup.submenu
  {
    iup.menu
    {
      iup.item{title="IupItem 1 Checked",value="ON"},
      iup.separator{},
      iup.item{title="IupItem 2 Disabled",active="NO"}
    }
    ;title="IupSubMenu 1"
  },
  iup.item{title="IupItem 3"},
  iup.item{title="IupItem 4"}
}:popup(iup.CENTER, iup.CENTER)
```

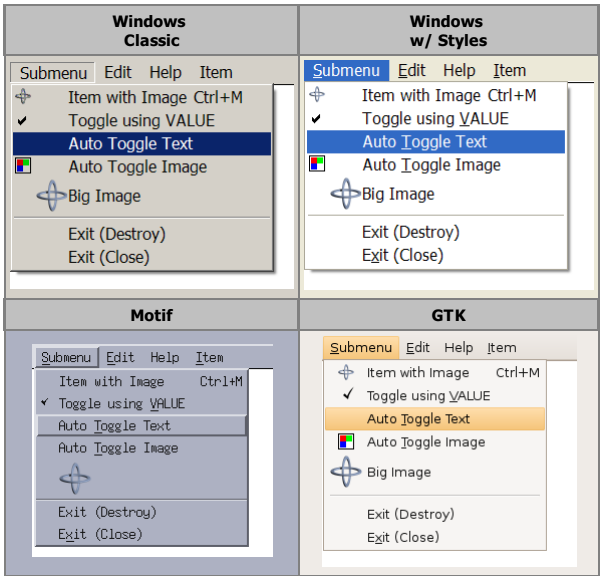
The same example using the cleaner syntax:

```
mnu = iup.menu
{
  {
    "IupSubMenu 1",
    iup.menu
    {
      {"IupItem 1 Checked";value="ON"},
      {},
      {"IupItem 2 Disabled";active="NO"}
    }
  },
  {"IupItem 3"},
  {"IupItem 4"}
}:popup(iup.CENTER, iup.CENTER)
```

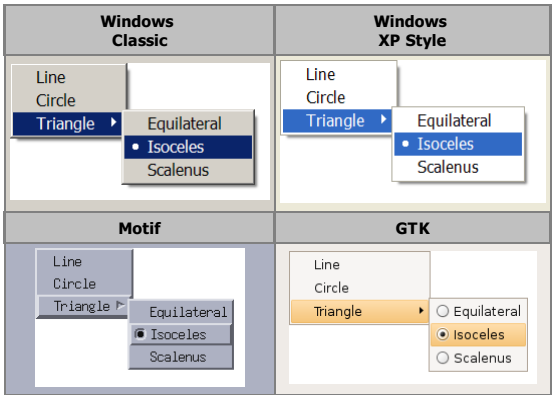
It is also possible to mix the cleaner syntax with the normal syntax or with already create elements.

Examples

Browse for Example Files



The **IupItem** check is affected by the RADIO attribute in **IupMenu**:



See Also

IupDialog, IupItem, IupSeparator, IupSubmenu, IupPopup, IupDestroy

IupSeparator

Creates the separator interface element. It shows a line between two menu items.

Creation

```
Ihandle* IupSeparator(void); [in C]
```

```
iup.separator{} -> (ih: ihandle) [in Lua]
separator() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Notes

The separator is ignored when it is part of the definition of the items in a bar menu.

Examples

Browse for Example Files

See Also

IupItem, IupSubMenu, IupMenu.

IupSubmenu

Creates a menu item that, when selected, opens another menu.

Creation

```
Ihandle* IupSubmenu(const char *title, Ihandle *menu); [in C]
iup.submenu{menu: ihandle[; title = title: string]} -> (ih: ihandle) [in Lua]
submenu(title, menu) [in LED]
```

title: String containing the text to be shown on the item. It can be NULL (nil in Lua), not optional in LED. It will set the TITLE attribute.
menu: optional child menu identifier. It can be NULL (nil in Lua), not optional in LED.

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

IMAGE [Windows and GTK Only] (non inheritable): Image name of the submenu image. In Windows, an item in a menu bar cannot have a check mark. Ignored if submenu in a menu bar. A recommended size would be 16x16 to fit the image in the menu item. In Windows, if larger than the check mark area it will be cropped. (since 3.0)

KEY (non inheritable): Underlines a key character in the submenu title. It is updated only when TITLE is updated. **Deprecated**, use the mnemonic support directly in the TITLE attribute.

TITLE (non inheritable): Submenu Text. The "&" character can be used to define a mnemonic, the next character will be used as key. Use "&&" to show the "&" character instead on defining a mnemonic.

WID (non inheritable): In Windows, returns the HMENU of the parent menu and it is actually created only when its child menu is mapped.

ACTIVE: also accepted.

Callbacks

HIGHLIGHT_CB: Action generated when the submenu is highlighted.

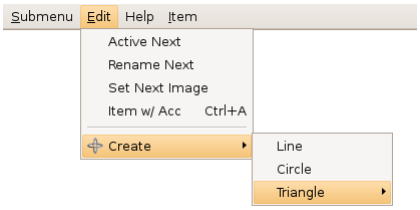
MAP_CB, UNMAP_CB, DESTROY_CB: common callbacks are supported.

Notes

In Motif and GTK, the text font will be affected by the dialog font when the menu is mapped.

Examples

Browse for Example Files



See the **IupMenu** element for more screenshots.

See Also

IupItem, IupSeparator, IupMenu.

KEY

Associates a key to a menu item or submenu. Such key works as a shortcut when the menu is open, this is not a hot key.

Value

String containing a key description. Its is a string representation of an IUP key code. Please refer to the Keyboard Codes table for a list of the possible values.

Default: NULL

Notes

IUP automatically underlines the first appearance of the chosen menu letter. For such, the chosen letter must necessarily be a part of the menu text.

In Windows, when used will also set an underscore on the respective letter of the submenu title.

The key will be used when navigating in the parent menu that contains the item. If the same character key is present in the title, then it will be underlined.

In the menu bar, some systems automatically associate the ALT+<letter> combination for the chosen letter. This is valid for the Windows driver, but not for the Motif driver.

Be careful not to misuse this attribute in relation to K_ANY or K_* callbacks.

Affects

IupItem, IupSubMenu.

HIGHLIGHT_CB

Callback triggered every time the user selects an **IupItem** or **IupSubmenu**.

Callback

```
int function(Ihandle *ih); [in C]
elem:highlight_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Affects

IupItem, IupSubmenu

OPEN_CB

Called just before the menu is opened.

Callback

```
int function(Ihandle *ih); [in C]
ih:open_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Affects

IupMenu

MENUCLOSE_CB

Called just after the menu is closed.

Callback

```
int function(Ihandle *ih); [in C]
ih:menuclose_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Affects

IupMenu

IupSetHandle

Associates a name with an interface element.

Parameters/Return

```
Ihandle *IupSetHandle(const char *name, Ihandle *ih); [in C]
iup.SetHandle(name: string, ih: ihandle) -> old_ih: ihandle [in Lua]
```

name: name of the interface element.
ih: identifier of the interface element. Use NULL to remove the association.

Returns: the identifier of the interface element previously associated to the parameter **name**.

Notes

This function is used so it is possible to set attributes values that are in fact other elements that were created in C. For example:

```
IupSetHandle("test_image", image);
IupSetAttribute(button, "IMAGE", "test_image");
```

But this code can be replaced by a more convenient function call:

```
IupSetAttributeHandle(button, "IMAGE", image);
```

In Lua this is not necessary, you can simply do:

```
button.image = image
```

that the association will be automatically made.

In fact, any pointer can be stored and retrieved with **IupSetHandle** and **IupGetHandle**, not only Ihandle*.

Also **IupSetHandle** can be called several times with the same pointer and different names. There is no restriction for the number of names a pointer can have, but **IupGetName** will return only the last name set.

When **IupSetHandle** is called, the control will have a HANDLENAME attribute with the last name set. (since 3.17)

See Also

IupGetHandle, IupSetAttributeHandle

IupGetHandle

Returns the identifier of an interface element that has an associated name using **IupSetHandle** or using LED.

Parameters/Return

```
Ihandle *IupGetHandle(const char *name); [in C]
iup.GetHandle(name: string) -> ih: ihandle [in Lua]
```

name: name of an interface element.

Returns: the element handle or NULL if not found.

Notes

This function is used for integrating IUP and LED. To manipulate an interface element defined in LED, first capture its identifier using function **IupGetHandle**, passing the name of the interface element as parameter, then use this identifier on the calls to IUP functions – for example, a call to the function that verifies the value of an attribute using **IupGetAttribute**.

See Also

IupSetHandle.

IupGetName

Returns a name of an interface element, if the element has an associated name using **IupSetHandle** or using LED (which calls **IupSetHandle** when parsed).

Notice that a handle can have many names. **IupGetName** will return the last name set.

Parameters/Return

```
char* IupGetName(Ihandle* ih); [in C]
iup.GetName(ih: ihandle) -> (name: string) [in Lua]
```

ih: Identifier of the interface element.

Returns: the name.

Notes

This name is not associated with the Lua variable name; this was inherited from LED and is needed for some functions.

See Also

IupSetHandle, IupGetHandle, IupGetAllNames.

IupGetAllNames

Returns the names of all interface elements that have an associated name using **IupSetHandle** or using LED.

Parameters/Return

```
int IupGetAllNames(char** names, int max_n); [in C]
iup.GetAllNames([max_n: number]) -> (names: table, n: number) [in Lua]
```

names: table receiving the names. Only the list of names need to be allocated. Each name will point to an internal string.
max_n: maximum number of names the table can receive. Can be omitted in Lua.

Returns: the number of names loaded to the table. If names==NULL or max_n==0 then returns the actual number of names.

Notes

This name is not associated to the name of the Lua variable – this was inherited from LED and is needed for some functions.

See Also

IupSetHandle, IupGetHandle, IupGetName, IupGetAllDialogs.

IupGetAllDialogs

Returns the names of all dialogs that have an associated name using **IupSetHandle** or using LED. Other dialogs will not be returned.

Parameters/Return

```
int IupGetAllDialogs(char** names, int max_n); [in C]
iup.GetAllDialogs([max_n: number]) -> (names: table, n: number) [in Lua]
```

names: table receiving the names. Only the list of names need to be allocated. Each name will point to an internal string.
max_n: maximum number of names the table can receive. Can be omitted in Lua.

Returns: the number of names loaded to the table. If names==NULL or max_n==0 then returns the actual number of names.

Notes

This name is not associated to the name of the Lua variable – this was inherited from LED and is needed for some functions.

See Also

IupSetHandle, IupGetHandle, IupGetName, IupGetAllNames.

IupSetLanguage

Sets the language name used by some pre-defined dialogs. Can also be changed using the global attribute LANGUAGE.

Parameters/Return

```
void IupSetLanguage(const char *name); [in C]

iup.SetLanguage(name: string) [in Lua]
```

name: Language name to be used. Can have one of the following values:

- "ENGLISH"
- "PORTUGUESE"

default: "ENGLISH".

Affects

All elements that have pre-defined texts. The pre-defined texts will be stored using IupSetLanguageString.

The native dialogs like **IupFileDlg** will always be displayed in the system language.

Even if the language is not supported (meaning its pack of pre-defined strings are not defined) the new language name will be successfully stored so you can set your own strings and return a coherent value, and the current defined string will not be changed.

Here is a list of the pre-defined string names:

```
IUP_ERROR               IUP_HELP
IUP_YES                 IUP_RED
IUP_NO                  IUP_GREEN
IUP_INVALIDDIR          IUP_BLUE
IUP_FILEISDIR           IUP_HUE
IUP_FILENOTEXIST        IUP_SATURATION
IUP_FILEOVERWRITE       IUP_INTENSITY
IUP_CREATEFOLDER        IUP_OPACITY
IUP_NAMENEWFOLDER       IUP_PALETTE
IUP_SAVEAS              IUP_TRUE
IUP_OPEN                IUP_FALSE
IUP_SELECTDIR           IUP_FAMILY
IUP_OK                  IUP_STYLE
IUP_CANCEL              IUP_SIZE
IUP_GETCOLOR            IUP_SAMPLE
```

Examples

```
#include "iup.h"

void main(void)
{
  IupOpen();
  IupSetLanguage("ENGLISH");
  IupMessage("IUP Language", IupGetLanguage());
  IupClose();
}
```

See Also

IupGetLanguage, IupSetLanguageString, LANGUAGE

IupGetLanguage

Returns the language used by some pre-defined dialogs. Returns the same value as the LANGUAGE global attribute.

Parameters/Return

```
char* IupGetLanguage(void); [in C]
iup.GetLanguage() -> (language: string) [in Lua]
```

Returns: the language.

See Also

IupSetLanguage, LANGUAGE.

IupSetLanguageString

Associates a name with a string as an auxiliary method for Internationalization of applications.

Parameters/Return

```
void IupSetLanguageString(const char *name, const char *value); [in C]
void IupStoreLanguageString(const char *name, const char *value);

iup.SetLanguageString(name, value: string) [in Lua]
```

name: name of the string.
value: string value.

Notes

IupStoreLanguageString will duplicate the string internally. **IupSetLanguageString** will store the pointer.

Elements that have pre-defined texts use this function when the current language is changed using **IupSetLanguage**.

IUP will **not** store strings for several languages at the same time, it will store only for the current language. When **IupSetLanguage** is called only the internal pre-defined strings are replace in the internal database. The application must register again all its strings for the new language.

If a dialog is created with string names associations and the associations are about to be changed, then the dialog must be destroyed **before** the associations are changed, then created again.

Associations are retrieved using the **IupGetLanguageString** function. But to simplify the usage of the string names associations attributes set with regular **IupSetStr*** functions can use the prefix "_@" to indicate a string name and not the actual string. This includes any attributes set in LED or in Lua. **IupSetAttribute*** functions can not be used because they simply store a pointer that may not be a string.

Examples

```
// If Language is Englih
IupSetLangueString("IUP_CANCEL", "Cancel");
  or
// If Language is Portuguese
IupSetLangueString("IUP_CANCEL", "Cancelar");


// Then when setting a button title use:
Ihandle* button_cancel = IupButton(IupGetLanguageString("IUP_CANCEL"), NULL);
  or
Ihandle* button_cancel = IupButton("_@IUP_CANCEL", NULL);
  or
IupSetStrAttribute(button_cancel, "TITLE", "_@IUP_CANCEL");
```

See Also

IupGetLanguageString, IupSetLanguagePack

IupGetLanguageString

Returns a language dependent string. The string must have been associated with the name using the **IupSetLanguageString** or **IupSetLanguagePack** functions.

Parameters/Return

```
char* IupGetLanguageString(const char* name); [in C]
iup.GetLanguageString(name: string) -> (value: string) [in Lua]
```

Returns: a string associated with the name.

Notes

If the association is not found returns the **name** itself.

See IupSetLanguageString for an example.

See Also

IupSetLanguageString, IupSetLanguagePack.

IupSetLanguagePack

Sets a pack of associations between names and string values. Internally will call **IupSetLanguageString** for each name in the pack.

Parameters/Return

```
void IupSetLanguagePack(Ihandle* ih); [in C]

iup.SetLanguagePack(ih: ihandle) [in Lua]
```

ih: pack of name-value association. It is simply a **IupUser** element with several attributes set.

Notes

After setting the pack it can be destroyed.

The existent associations will not be removed. But if the new ones have the same names, the old ones will be replaced. If set to NULL will remove all current associations.

Examples

```
 pack = iup.user{}
pack["IUP_RED"] = "Vermelho"
pack["MY_ITEMCOLORTEST"] = "Teste de Cor"
iup.SetLanguagePack(pack)
iup.Destroy(pack)
```

See Also

IupGetLanguageString, IupSetLanguageString, IupUser

IupClipboard (since 3.0)

Creates an element that allows access to the clipboard. Each clipboard should be destroyed using IupDestroy, but you can use only one for the entire application because it does not store any data inside. Or you can simply create and destroy every time you need to copy or paste.

Creation

```
Ihandle* IupClipboard(void); [in C]
iup.clipboard{} -> (ih: ihandle) [in Lua]
clipboard() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

ADDFORMAT (write-only): register a custom format for clipboard data given its name. The registration remains valid even after the element is destroyed. A new format must be added before used. (since 3.7)

EMFAVAILABLE (read-only) [Windows Only]: informs if there is a Windows Enhanced Metafile available at the clipboard. (Since 3.2)

FORMAT: set the current format to be used by the FORMATAVAILABLE and FORMATDATA attributes. (since 3.7)

FORMATAVAILABLE (read-only): informs if there is a data in the FORMAT available at the clipboard. If FORMAT is not set returns NULL. (since 3.7)

FORMATDATA: sets or retrieves the data from the clipboard in the format defined by the FORMAT attribute. If FORMAT is not set returns NULL. If set to NULL clears the clipboard data. When set the FORMATDATASIZE attribute must be set before with the data size. When retrieved FORMATDATASIZE will be set and available after data is retrieved. (since 3.7)

FORMATDATASIZE: size of the data on the clipboard. Used by the FORMATDATA attribute processing. (since 3.7)

IMAGE (write-only): name of an image to copy to the clipboard. If set to NULL clears the clipboard data. (GTK 2.6)

IMAGEAVAILABLE (read-only): informs if there is an image available at the clipboard. (GTK 2.6)

NATIVEIMAGE: native handle of an image to copy or paste, to or from the clipboard. In Win32 is a **HANDLE** of a DIB. In GTK is a **GdkPixbuf***. In Motif is a **Pixmap**. If set to NULL clears the clipboard data. The returned handle in a paste must be released after used (GlobalFree(handle), g_object_unref(pixbuf) or XFreePixmap(display, pixmap)). After copy, do NOT release the given handle. See IUP-IM Functions for utility functions on image native handles. (GTK 2.6)

SAVEEMF (write-only) [Windows Only]: saves the EMF from the clipboard to the given filename. (Since 3.2)

SAVEWMF (write-only) [Windows Only]: saves the WMF from the clipboard to the given filename. (Since 3.2)

TEXT: copy or paste text to or from the clipboard. If set to NULL clears the clipboard data.

TEXTAVAILABLE (read-only): informs if there is a text available at the clipboard.

WMFAVAILABLE (read-only) [Windows Only]: informs if there is a Windows Metafile available at the clipboard. (Since 3.2)

Notes

In Windows when "TEXT" format data is copied to the clipboard, the system will automatically store other text formats too if those formats are not already stored. This means that when copying "TEXT" Windows will also store "Unicode Text" and "OEM Text", but only if those format were not copied before. So to make sure the system will copy all the other text formats clear the clipboard before copying you own data (you can simply set TEXT=NULL before setting the actual value).

Examples

```
Ihandle* clipboard = IupClipboard();
IupSetAttribute(clipboard, "TEXT", IupGetAttribute(text, "VALUE"));
IupDestroy(clipboard);
```

```
Ihandle* clipboard = IupClipboard();
IupSetAttribute(text, "VALUE", IupGetAttribute(clipboard, "TEXT"));
IupDestroy(clipboard);
```

IupTimer

Creates a timer which periodically invokes a callback when the time is up. Each timer should be destroyed using IupDestroy.

Creation

```
Ihandle* IupTimer(void); [in C]
iup.timer{} -> (ih: ihandle) [in Lua]
timer() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

TIME: The time interval in milliseconds. In Windows the minimum value is 10ms.

RUN: Starts and stops the timer. Possible values: "YES" or "NO". Returns the current timer state. If you have multiple threads start the timer in the main thread.

WID (read-only): Returns the native serial number of the timer. Returns -1 if not running. A timer is mapped only when it is running.

Callbacks

ACTION_CB: Called every time the defined time interval is reached. To stop the callback from being called simply stop de timer with RUN=NO. Inside the callback the attribute ELAPSEDTIME returns the time elapsed since the timer was started (since 3.15).

```
int function(Ihandle *ih); [in C]
ih:action_cb() -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event.

Returns: IUP_CLOSE will be processed.

Examples

Browse for Example Files

IupTuioClient (since 3.3)

Implements a TUIO protocol client that allows the use of multi-touch devices. It can use any TUIO server, but it was tested with the Community Core Vision (CCV) from the NUI Group.

Initialization and usage

The **IupTuioOpen** function must be called after a **IupOpen**, so that the control can be used. The iuptuio.h file must also be included in the source code. The program must be linked to the controls library (iuptuio). There is no external dependencies, the TUIO client library is already included.

To make the control available in Lua use require"iupluatuio" or manually call the initialization function in C, **iuptuiolua_open**, after calling **iuplua_open**. When manually calling the function the iupluatuio.h file must also be included in the source code and the program must be linked to the respective Lua control library (iupluatuio).

Creation

```
Ihandle* IupTuioClient(int port); [in C]
iup.tuioclient{[port: number]} -> (ih: ihandle) [in Lua]
tuioclient(port) [in LED]
```

port: the UDP port used to connect to the TUIO server. If 0 is specified then the default value of 3333 will be used (in Lua it can be simply omitted).

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

CONNECT: connects (YES) or disconnects (NO) to the TUIO server. Returns the connected state. If LOCKED is used when connected the **IupSetAttribute** will not return until it is disconnected (not recommended).

DEBUG: when set will enable a print a log of TUIO cursor messages on standard output.

TARGETCANVAS: name of a handle to an **IupCanvas** that will be used to receive the events.

Callbacks

TOUCH_CB: Action generated when a touch event occurred. Multiple touch events will trigger several calls.

```
int function(Ihandle* ih, int id, int x, int y, char* state); [in C]
ih:touch_cb(id, x, y: number, state: string) -> (ret: number) [in Lua]
```

ih: identifies the element that activated the event. If TARGETCANVAS is not defined then it is the **IupTuioClient** control.
id: identifies the touch point.
x, **y**: position in pixels, relative to the top-left corner of the canvas, or the screen if TARGETCANVAS is not defined.
state: the touch point state. Can be: DOWN, MOVE or UP. If the point is a "primary" point then "-PRIMARY" is appended to the string.

Returns: IUP_CLOSE will be processed.

MULTITOUCH_CB: Action generated when multiple touch events occurred.

```
int function(Ihandle *ih, int count, int* pid, int* px, int* py, int* pstate) [in C]
ih:multitouch_cb(count: number, pid, px, py, pstate: table) -> (ret: number) [in Lua]
```

ih: identifier of the element that activated the event. If TARGETCANVAS is not defined then it is the **IupTuioClient** control.
count: Number of touch points in the array.
pid: Array of touch point ids.
px: Array of touch point x coordinates in pixels, relative to the top-left corner of the canvas, or the screen if TARGETCANVAS is not defined.
py: Array of touch point y coordinates in pixels, relative to the top-left corner of the canvas, or the screen if TARGETCANVAS is not defined.
pstate: Array of touch point states. Can be 'D' (DOWN), 'U' (UP) or 'M' (MOVE).

Returns: IUP_CLOSE will be processed.

Notes

The cursor ID used in the callbacks is the session ID. In TUIO when a cursor is destroyed another cursor can be created with the same ID, the difference between them is the session ID that is always incremented every time a cursor is added or removed. We consider the primary cursor the existing cursor with the smaller session ID.

The native support for multi-touch in Windows 7 uses the same callbacks described here without the need of a **IupTuioClient** control. So the application will work without change. But the attribute TOUCH=YES must be set on the **IupCanvas**, and coordinates will be always relative to the top-left corner of the canvas.

The **IupTuioClient** does not emulates a mouse for single touch events. But as you can see from the example a mouse emulator can be easily implemented.

To learn more about TUIO:

http://www.tuio.org

Examples

Browse for Example Files (see canvas1.c)

IupUser

Creates a user element in IUP, which is not associated to any interface element. It is used to map an external element to a IUP element. Its use is usually for additional elements, but you can use it to create an Ihandle* to store private attributes.

It is also a void container. Its children can be dynamically added using IupAppend or IupInsert.

Creation

```
Ihandle* IupUser(void); [in C]
iup.user{} -> (ih: ihandle) [in Lua]
user() [in LED]
```

Returns: the identifier of the created element, or NULL if an error occurs.

Attributes

CLEARATTRIBUTES (write-only, non inheritable): it will clear all attributes stored internally and remove all references. (since 3.0)

IupConfig (since 3.12)

A group of functions to load, store and save application configuration variables. For example, the list of Recent Files, the last position and size of a dialog, last used parameters in dialogs, etc.

To use the functions in C/C++ you must include the "iup_config.h" header. The functions are NOT available in LED. Binding Lua since 3.15.

Each variable has a key name, a value and a group that it belongs to. The file is based on a simple configuration file like ".ini" or ".cfg". Each ground can has more than one key, but all keys in the same group must have different names. Group and Key names can NOT have a period ".". The file syntax is such as:

```
[Group]
Key=Value
Key=Value
...
```

Guide

First create a new configuration database using the **IupConfig** constructor. To destroy it use the **IupDestroy** function. Then when the application is started call **IupConfigLoad** and when the application is about to close, call **IupConfigSave**.

To retrieve variables use the **IupConfigGetVariable*** functions and after they where changed store them using the **IupConfigSetVariable*** functions.

Creation

```
Ihandle* IupConfig(void); [in C]
iup.config{}-> ih: ihandle [in Lua]

[not available in LED]
```

Returns a new database where the variables will be stored.

File Storage

```
int IupConfigLoad(Ihandle* ih); [in C]
iup.ConfigLoad(ih: ihandle) -> (ret: number) [in Lua]
or ih:Load() -> (ret: number) [in Lua]

int IupConfigSave(Ihandle* ih); [in C]
iup.ConfigSave(ih: ihandle) -> (ret: number) [in Lua]
or ih:Save() -> (ret: number) [in Lua]
```

ih: Identifier of the configuration database

Returns: an error code. 0= no error; -1=error opening the file; -2=error accessing the file; -3=error during filename construction

Loads or saves the configuration file.

The filename (with path) can be set using a regular attribute called APP_FILENAME.

But the most interesting is to let the filename to be dynamically constructed using the APP_NAME attribute. In this case APP_FILENAME must **not** be defined. The file name creation will depend on the system and on its usage.

There are two defined usages. First, for a **User Configuration File,** it will be stored on the user Home folder. Second, as an **Application Configuration File,** it will be stored in the same folder of the executable.

The **User Configuration File** is the most common usage. In UNIX, the filename will be "<HOME>/.<APP_NAME>", where "<HOME>" is replaced by the "HOME" environment variable contents, and <APP_NAME> replaced by the APP_NAME attribute value. In Windows, the filename will be "<HOMEDRIVE><HOMEPATH>\<APP_NAME>.cfg", where HOMEDRIVE and HOMEPATH are also obtained from environment variables.

The **Application Configuration File** is defined by setting the attribute APP_CONFIG to Yes (default is No). In this case the attribute APP_PATH must also be set. In UNIX, the filename will be "<APP_PATH>.<APP_NAME>", and in Windows will be "<APP_PATH><APP_NAME>.cfg". Notice that the attribute APP_PATH must contains a folder separator "/" at the end.

After the functions are called the attribute FILENAME is set reflecting the constructed filename.

So usually at start up, an application will do:

```
Ihandle* config = IupConfig();
IupSetAttribute(config, "APP_NAME", "MyAppName");
IupConfigLoad(config);
```

Variables

```
void IupConfigSetVariableStr(Ihandle* ih, const char* group, const char* key, const char* value);  [in C]
void IupConfigSetVariableStrId(Ihandle* ih, const char* group, const char* key, int id, const char* value);
void IupConfigSetVariableInt(Ihandle* ih, const char* group, const char* key, int value);
void IupConfigSetVariableIntId(Ihandle* ih, const char* group, const char* key, int id, int value);
void IupConfigSetVariableDouble(Ihandle* ih, const char* group, const char* key, double value);
void IupConfigSetVariableDoubleId(Ihandle* ih, const char* group, const char* key, int id, double value);
iup.ConfigSetVariable(ih: ihandle, group, key: string, value: string or number)  [in Lua]
or ih:SetVariable(group, key: string, value: string or number)  [in Lua]
iup.ConfigSetVariable(ih: ihandle, group, key: string, id: number, value: string or number)  [in Lua]
or ih:SetVariable(group, key: string, id: number, value: string or number)  [in Lua]

const char* IupConfigGetVariableStr(Ihandle* ih, const char* group, const char* key);  [in C]
const char* IupConfigGetVariableStrId(Ihandle* ih, const char* group, const char* key, int id);
int    IupConfigGetVariableInt(Ihandle* ih, const char* group, const char* key);
int    IupConfigGetVariableIntId(Ihandle* ih, const char* group, const char* key, int id);
double IupConfigGetVariableDouble(Ihandle* ih, const char* group, const char* key);
double IupConfigGetVariableDoubleId(Ihandle* ih, const char* group, const char* key, int id);
iup.ConfigGetVariable(ih: ihandle, group, key: string) -> (value: string) [in Lua]
or ih:GetVariable(ih: ihandle, group, key: string) -> (value: string) [in Lua]
iup.ConfigGetVariable(ih: ihandle, group, key: string, id: number) -> (value: string) [in Lua]
or ih:GetVariable(ih: ihandle, group, key: string, id: number) -> (value: string) [in Lua]

const char* IupConfigGetVariableStrDef(Ihandle* ih, const char* group, const char* key, const char* def);  [in C]
const char* IupConfigGetVariableStrIdDef(Ihandle* ih, const char* group, const char* key, int id, const char* def);
int    IupConfigGetVariableIntDef(Ihandle* ih, const char* group, const char* key, int def);
int    IupConfigGetVariableIntIdDef(Ihandle* ih, const char* group, const char* key, int id, int def);
double IupConfigGetVariableDoubleDef(Ihandle* ih, const char* group, const char* key, double def);
double IupConfigGetVariableDoubleIdDef(Ihandle* ih, const char* group, const char* key, int id, double def);
iup.ConfigGetVariableDef(ih: ihandle, group, key: string, def: string or number) -> (value: string) [in Lua]
or ih:GetVariableDef(group, key: string, def: string or number) -> (value: string) [in Lua]
iup.ConfigGetVariableDef(ih: ihandle, group, key: string, id: number, def: string or number) -> (value: string) [in Lua]
or ih:GetVariableDef(group, key: string, id: number, def: string or number) -> (value: string) [in Lua]
```

ih: Identifier of the configuration database
group: group name of the variable
key: key name of the variable
id: used when the variable has a sequential number
value: value of the variable
def: default value of the variable

Returns: the variable value or NULL (or 0 for integer and double) if the variable is not set or does not exist. When the variable may not exist you can use the functions with **def** to use a default value.

These functions are very similar to the **IupSetAttribute** and **IupGetAttribute** functions. Internally the variables are stored as attributes using a "<GROUP>.<KEY>" combination, that's why group and key names can not have periods ".".

Recent File Menu

```
void IupConfigRecentInit(Ihandle* ih, Ihandle* menu, Icallback recent_cb, int max_recent);
iup.ConfigRecentInit(ih, menu: ihandle, max_recent: number)  [in Lua]
or ih:RecentInit(menu: ihandle, max_recent: number)  [in Lua]
ih:recent_cb() -> (ret: number) [in Lua]

void IupConfigRecentUpdate(Ihandle* ih, const char* filename);
iup.ConfigRecentUpdate(ih: ihandle, filename: string)  [in Lua]
or ih:RecentUpdate(filename: string)  [in Lua]
```

ih: Identifier of the configuration database
menu: menu where the recent file items will be listed
recent_cb: callback that will be called when a recent file item is selected on the menu
max_recent: the maximum number of recent file items.
filename: name of the file that where just saved or open

These functions store and manage a "Recent Files" menu for the application. Call **IupConfigRecentInit** once to initialize the menu. Then every time a file is open or saved call **IupConfigRecentUpdate** so that the menu list is updated. The last file will be always on the top of the list.

Inside the RECENT_CB callback the TITLE attribute contains the filename, but the ih handle is not the menu, it is the IupConfig handle. But also inside the callback the IupConfig will inherit attributes from the menu as if it was its parent. (since 3.15)

Dialog Position and Size

```
void IupConfigDialogShow(Ihandle* ih, Ihandle* dialog, const char* name);
iup.ConfigDialogShow(ih, dialog: ihandle, name: string)  [in Lua]
or ih:DialogShow(dialog: ihandle, name: string)  [in Lua]

void IupConfigDialogClosed(Ihandle* ih, Ihandle* dialog, const char* name);
iup.ConfigDialogClosed(ih, dialog: ihandle, name: string)  [in Lua]
or ih:DialogClosed(dialog: ihandle, name: string)  [in Lua]
```

ih: Identifier of the configuration database
dialog: the dialog to manage the size and position
name: a name for this dialog

These functions store and manage the position and size of a dialog. So when the application is run again the dialog can be show at its last position and last size. Use the function **IupConfigDialogShow** to show the dialog adjusting its size and position. And use the function **IupConfigDialogClosed** to save the last dialog position and size when the dialog is about to be closed, usually inside the dialog CLOSE_CB callback.

IupConfigDialogShow does no adjustments if the dialog is already visible, just call **IupShow**. If the dialog was closed maximized it will be shown maximized. The default size, at the first time ever the dialog is shown, is maximized. The dialog size is set only if RESIZE=Yes. (since 3.16)

The position is saved in the variables "X" and "Y" of the given group name. The size is saved in the variables "Width" and "Height" of the given group name.

If your dialog is resizable and you want to avoid the last size usage because you changed the dialog layout, then reset the "Width" and "Height" variables before calling **IupConfigDialogShow**.

To avoid the dialog size to be maximized, set the variable "Maximized" to 0 before calling **IupConfigDialogShow**. (since 3.16)

To use **IupConfigDialogShow** for a modal dialog, call it before calling **IupPopup** with IUP_CURRENT. (since 3.16)

See Also

IupDestroy, IupSetAttribute, IupGetAttribute

IupExecute (since 3.17)

Runs the executable with the given parameters.

It is a non synchronous operation, i.e. the function will return just after execute the command and it will not wait for its result.

In Windows, there is no need to add the ".exe" file extension.

Used by the IupHelp function.

Parameters/Return

```
int IupExecute(const char* filename, const char* parameters); [in C]
iup.Execute(filename[, parameters]: string) -> (ret: number) [in Lua]
```

filename: name of the executable. Can contains a path.
parameters: optional parameters. Can be NULL.

Returns: 1 if successful, -1 if failed. In Windows can return -2 if file not found.

IupHelp

Opens the given URL. In UNIX executes Netscape, Safari (MacOS) or Firefox (in Linux) passing the desired URL as a parameter. In Windows executes the shell "open" operation on the given URL.

In UNIX you can change the used browser setting the environment variable IUP_HELPAPP or using the global attribute "HELPAPP".

It is a non synchronous operation, i.e. the function will return just after execute the command and it will not wait for its result.

Since IUP 3.17, it will use the IupExecute function.

Parameters/Return

```
int IupHelp(const char* url); [in C]
iup.Help(url: string) -> (ret: number) [in Lua]
```

url: may be any kind of address accepted by the Browser, that is, it can include 'http://', or be just a file name, etc.

Returns: 1 if successful, -1 if failed. In Windows can return -2 if file not found.

iupMask (deprecated since 3.0, will be removed in a future version)

Since IUP 3.0, IupText and IupMatrix have several MASK* attributes to support masks. See the MASK attribute for more information.

Functions to associate an input mask to a **IupText** or a **IupMatrix** element.

These functions are included in the IupControls library.

Functions

```
int iupMaskSet(Ihandle *ih, const char *mask, int autofill, int casei);
int iupMaskMatSet(Ihandle *ih, const char *mask, int autofill, int casei, int lin, int col);
```

These functions are responsible for setting the mask to be used.

ih: Ihandle of IupText or IupMatrix
mask: Mask to be used
autofill: When 1, turns the auto-fill mode on. In auto-fill mode, whenever possible, literal characters will be automatically added to the field (NOT supported since 3.0)
casei: When 1, uppercase or lowercase characters will be treated indistinctly
lin, col: Line and column numbers in the matrix

They return 1 if the mask is set, or 0 if there is an error (e.g., invalid mask).

```
int iupMaskSetInt(Ihandle *ih, int autofill, int min, int max);
int iupMaskSetFloat(Ihandle *ih, int autofill, float min, float max);
int iupMaskMatSetInt(Ihandle *ih, int autofill, int min, int   max, int lin, int col);
int iupMaskMatSetFloat(Ihandle *ih, int autofill, float min, float max, int lin, int col);
```

These functions set a mask that defines a limit to a number. Works only for integers and floats. Limitations: since the validation process is performed key by key, the user cannot type intermediate numbers (even inside the limit) if they are not following the mask rules.

ih: Ihandle of IupText or IupMatrix
autofill: When 1, turns the auto-fill mode on. In auto-fill mode, whenever possible, literal characters will be automatically added to the field (NOT supported since 3.0)
min: Minimum value accepted in the field
max: Maximum value accepted in the field
lin, col: Line and column numbers in the matrix

They always return 1.

```
int iupMaskRemove(Ihandle *ih);
int iupMaskMatRemove(Ihandle *ih, int lin, int col);
```

These functions are responsible for removing the mask from the control.

ih: Ihandle of IupText or IupMatrix
lin, col: Line and column numbers in the matrix

```
int iupMaskCheck (Ihandle *ih);
int iupMaskMatCheck(Ihandle *ih, int lin, int col);
```

These functions verify if what was typed by the user is valid for the defined mask. For IupMatrix they work only when in edition mode.

ih: Ihandle of IupText or IupMatrix
lin, col: Line and column numbers in the matrix

They return 1 if the text satisfies the mask, or 0 otherwise.

```
int iupMaskGet(Ihandle *ih, char **val);
int iupMaskGetFloat(Ihandle *ih, float *fval);
int iupMaskGetInt(Ihandle *ih, int *ival);
int iupMaskMatGet(Ihandle *ih, char **val, int lin, int col);
int iupMaskMatGetFloat(Ihandle *ih, float  *fval, int lin, int col);
int iupMaskMatGetDouble(Ihandle *ih, double *dval, int lin, int col);
int iupMaskMatGetInt(Ihandle *ih, int *ival, int lin, int col);
```

These functions check if the text satisfies the mask, and they retrieve the fields value in only one call. For IupMatrix they work only when in edition mode.

ih: Ihandle of IupText or IupMatrix
val, fval, ival: Pointers used to complete the return value
lin, col: Line and column numbers in the matrix.

They return 1 if the text satisfies the mask, or 0 otherwise.

Notes

To make the use of masks simpler, the following predefined masks were created:

IUPMASK_FLOAT - Float number
IUPMASK_UFLOAT - Unsigned Float number
IUPMASK_EFLOAT - Float number with exponential notation
IUPMASK_INT - Integer number
IUPMASK_UINT - Unsigned Integer number

Examples

Browse for Example Files

SDK

Introduction

Internal SDK documentation of the IUP library, automatically generated using Doxygen (http://www.doxygen.org/).

Code Standards

Function Names (prefix format)

- IupFunc - User API, implemented in the core
- iupFunc - Internal Core API, implemented in the core, used in the core or in driver
- iupxxxFunc - Windows Internal API, implemented in driver xxx, used in driver xxx
- iupdrvFunc - Driver API, implemented in driver, used in the core or driver

- xxxFunc - Driver xxx local functions

Global Variables (lower case format)

- iupxxx_var

Local Variables (lower case format, using module name)

- iyyy_var

File Names

- iupyyy.h - public headers
- iup_yyy.h/c - core
- iupxxx_yyy.h/c - driver

Structures

- Iyyy

File Comments (at start)

- Check an existent file for example.

Defines

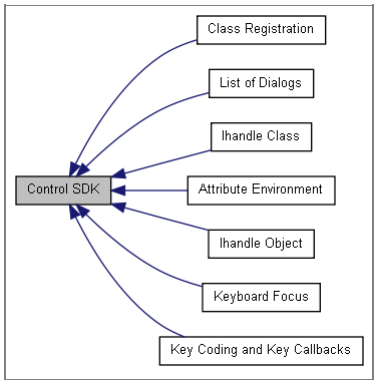
- __IUPXXX_H (for include file, same file name, upper case, "__" prefix and replace "." by "_")
- IUP_XXX (for global enumerations)
- IXXX_YYY (for local enumerations)
- iupXXX (for macros, complement with Function Names rules)

Documentation

- In the header, using Doxygen commands.
- Check an existent header for example.

Control SDK

Collaboration diagram for Control SDK:



Modules

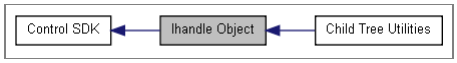
- Attribute Environment
- Ihandle Class
- List of Dialogs
- Keyboard Focus
- Key Coding and Key Callbacks
- Ihandle Object
- Class Registration

Detailed Description

Control Creation Guide

Ihandle Object
[Control SDK]

Collaboration diagram for Ihandle Object:



Data Structures

struct	Ihandle_

Modules

	Child Tree Utilities

Defines

#define	IUP_EXPAND_WIDTH (IUP_EXPAND_W1 | IUP_EXPAND_W0)
#define	IUP_EXPAND_HEIGHT (IUP_EXPAND_H1 | IUP_EXPAND_H0)
#define	IUP_EXPAND_BOTH (IUP_EXPAND_WIDTH | IUP_EXPAND_HEIGHT)
#define	iupALLOCCTRLDATA() ((IcontrolData*)calloc(1, sizeof(IcontrolData)))

Typedefs

typedef struct _InativeHandle	InativeHandle
typedef struct _IcontrolData	IcontrolData

Enumerations

enum	Iexpand {
	IUP_EXPAND_NONE = 0x00, **IUP_EXPAND_H0** = 0x01, **IUP_EXPAND_H1** = 0x02, **IUP_EXPAND_W0** = 0x04,
	IUP_EXPAND_W1 = 0x08, **IUP_EXPAND_HFREE** = 0x10, **IUP_EXPAND_WFREE** = 0x20
	}

enum	Iflags {
	IUP_FLOATING = 0x01, IUP_FLOATING_IGNORE = 0x02, IUP_MAXSIZE = 0x04, IUP_MINSIZE = 0x08,
	IUP_INTERNAL = 0x10
	}

Functions

void **	iupObjectGetParamList (void *first, va_list arglist)
int	iupObjectCheck (Ihandle *ih)

Detailed Description

Object handle for all the elements.

See iup_object.h

Define Documentation

#define IUP_EXPAND_WIDTH (IUP_EXPAND_W1 | IUP_EXPAND_W0)

Expand configuration

#define IUP_EXPAND_HEIGHT (IUP_EXPAND_H1 | IUP_EXPAND_H0)

Expand configuration

#define IUP_EXPAND_BOTH (IUP_EXPAND_WIDTH | IUP_EXPAND_HEIGHT)

Expand configuration

#define iupALLOCCTRLDATA ( ) ((IcontrolData*)calloc(1, sizeof(IcontrolData)))

IcontrolData allocation utility.

Typedef Documentation

typedef struct _InativeHandle InativeHandle

A simple definition that do not depends on the native system, but helps a lot when writing native code. See iup_object.h for definitions.

typedef struct _IcontrolData IcontrolData

Each control may define its own structure in its private module.

Enumeration Type Documentation

enum Iexpand

Expand configuration

enum Iflags

General flags.

Enumerator:

IUP_FLOATING	is a floating element. FLOATING=Yes

IUP_FLOATING_IGNORE	is a floating element. FLOATING=Ignore. Do not compute layout.
IUP_MAXSIZE	has the MAXSIZE attribute set
IUP_MINSIZE	has the MAXSIZE attribute set
IUP_INTERNAL	it is an internal element of the container

Function Documentation

void** iupObjectGetParamList	(	void *	*first*,		
		va_list	*arglist*		
	)				

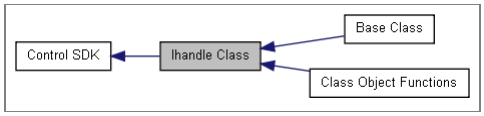
Utility that returns an array of parameters. Must call free for the returned value after usage. Used by the creation functions of objects that receives a NULL terminated array of parameters.

int iupObjectCheck	(	Ihandle *	*ih*	)	

Checks if the handle is still valid based on the signature. But if the handle was destroyed still can access invalid memory.

Ihandle Class
[Control SDK]

Collaboration diagram for Ihandle Class:



Data Structures

struct	Iclass_

Modules

	Class Object Functions
	Base Class

Typedefs

typedef enum _InativeType	InativeType
typedef enum _IchildType	IchildType
typedef char *(*	IattribGetFunc)(Ihandle *ih)
typedef char *(*	IattribGetIdFunc)(Ihandle *ih, int id)
typedef char *(*	IattribGetId2Func)(Ihandle *ih, int id1, int id2)
typedef int(*	IattribSetFunc)(Ihandle *ih, const char *value)
typedef int(*	IattribSetIdFunc)(Ihandle *ih, int id, const char *value)
typedef int(*	IattribSetId2Func)(Ihandle *ih, int id1, int id2, const char *value)
typedef enum _IattribFlags	IattribFlags

Enumerations

enum	_InativeType {
	IUP_TYPEVOID, IUP_TYPECONTROL, IUP_TYPECANVAS, IUP_TYPEDIALOG,
	IUP_TYPEIMAGE, IUP_TYPEMENU
	}

enum	_IchildType { IUP_CHILDNONE, IUP_CHILDMANY }

enum	_IattribFlags {
	IUPAF_DEFAULT = 0, IUPAF_NO_INHERIT = 1, IUPAF_NO_DEFAULTVALUE = 2, IUPAF_NO_STRING = 4,
	IUPAF_NOT_MAPPED = 8, IUPAF_HAS_ID = 16, IUPAF_READONLY = 32, IUPAF_WRITEONLY = 64,
	IUPAF_HAS_ID2 = 128, IUPAF_CALLBACK = 256, IUPAF_NO_SAVE = 512, IUPAF_NOT_SUPPORTED = 1024,
	IUPAF_IHANDLENAME = 2048, IUPAF_IHANDLE = 4096
	}

Functions

Iclass *	iupClassNew (Iclass *ic_parent)
void	iupClassRelease (Iclass *ic)
int	iupClassMatch (Iclass *ic, const char *classname)
void	iupClassRegisterAttribute (Iclass *ic, const char *name, IattribGetFunc get, IattribSetFunc set, const char *default_value, const char *system_default, int flags)

void	iupClassRegisterAttributeId (Iclass *ic, const char *name, IattribGetIdFunc get, IattribSetIdFunc set, int flags)
void	iupClassRegisterAttributeId2 (Iclass *ic, const char *name, IattribGetId2Func get, IattribSetId2Func set, int flags)
void	iupClassRegisterGetAttribute (Iclass *ic, const char *name, IattribGetFunc *get, IattribSetFunc *set, const char **default_value, const char **system_default, int *flags)
void	iupClassRegisterReplaceAttribFunc (Iclass *ic, const char *name, IattribGetFunc _get, IattribSetFunc _set)
void	iupClassRegisterReplaceAttribDef (Iclass *ic, const char *name, const char *_default_value, const char *_system_default)
void	iupClassRegisterReplaceAttribFlags (Iclass *ic, const char *name, int _flags)
void	iupClassRegisterCallback (Iclass *ic, const char *name, const char *format)
char *	iupClassCallbackGetFormat (Iclass *ic, const char *name)

Detailed Description

See iup_class.h

Typedef Documentation

typedef enum _InativeType InativeType

Known native types.

typedef enum _IchildType IchildType

Possible number of children.

typedef char*(* IattribGetFunc)(Ihandle *ih)

GetAttribute called for a specific attribute. Used by iupClassRegisterAttribute.

typedef char*(* IattribGetIdFunc)(Ihandle *ih, int id)

GetAttribute called for a specific attribute when has_attrib_id is 1.
Same as IattribGetFunc but handle attribute names with number ids at the end.
When calling iupClassRegisterAttribute just use a typecast.
-1 is used for invalid ids.
Pure numbers are translated into IDVALUEid. Used by iupClassRegisterAttribute.

typedef char*(* IattribGetId2Func)(Ihandle *ih, int id1, int id2)

GetAttribute called for a specific attribute when has_attrib_id is 1.
Same as IattribGetFunc but handle attribute names with number ids at the end.
When calling iupClassRegisterAttribute just use a typecast.
-1 is used for invalid ids.
Pure numbers are translated into IDVALUEid. Used by iupClassRegisterAttribute.

typedef int(* IattribSetFunc)(Ihandle *ih, const char *value)

SetAttribute called for a specific attribute.
If returns 0, the attribute will not be stored in the hash table (except inheritble attributes that are always stored in the hash table).
When IupSetAttribute is called using value=NULL, the default_value is passed to this function. Used by iupClassRegisterAttribute.

typedef int(* IattribSetIdFunc)(Ihandle *ih, int id, const char *value)

SetAttribute called for a specific attribute when has_attrib_id is 1.
Same as IattribSetFunc but handle attribute names with number ids at the end.
When calling iupClassRegisterAttribute just use a typecast.
-1 is used for invalid ids.
Pure numbers are translated into IDVALUEid, ex: "1" = "IDVALUE1". Used by iupClassRegisterAttribute.

typedef int(* IattribSetId2Func)(Ihandle *ih, int id1, int id2, const char *value)

SetAttribute called for a specific attribute when has_attrib_id is 2.
Same as IattribSetFunc but handle attribute names with number ids at the end.
When calling iupClassRegisterAttribute just use a typecast.
-1 is used for invalid ids.
Pure numbers are translated into IDVALUEid, ex: "1" = "IDVALUE1". Used by iupClassRegisterAttribute.

typedef enum _IattribFlags IattribFlags

Attribute flags. Used by iupClassRegisterAttribute.

Enumeration Type Documentation

enum _InativeType

Known native types.

Enumerator:

IUP_TYPEVOID	No native representation - HBOX, VBOX, ZBOX, FILL, RADIO (handle==(void*)-1 always)
IUP_TYPECONTROL	Native controls - BUTTON, LABEL, TOGGLE, LIST, TEXT, MULTILINE, FRAME, others
IUP_TYPECANVAS	Drawing canvas, also used as a base control for custom controls.
IUP_TYPEDIALOG	DIALOG
IUP_TYPEIMAGE	IMAGE

IUP_TYPEMENU	MENU, SUBMENU, ITEM, SEPARATOR

enum _IchildType

Possible number of children.

Enumerator:

IUP_CHILDNONE	can not add children using Append/Insert
IUP_CHILDMANY	can add any number of children. /n IUP_CHILDMANY+n can add n children.

enum _IattribFlags

Attribute flags. Used by iupClassRegisterAttribute.

Enumerator:

IUPAF_DEFAULT	inheritable, can has a default value, is a string, can call the set/get functions only if mapped, no ID
IUPAF_NO_INHERIT	is not inheritable
IUPAF_NO_DEFAULTVALUE	can not has a default value
IUPAF_NO_STRING	is not a string
IUPAF_NOT_MAPPED	will call the set/get functions also when not mapped
IUPAF_HAS_ID	can has an ID at the end of the name, automatically set by iupClassRegisterAttributeId
IUPAF_READONLY	is read-only, can not be changed
IUPAF_WRITEONLY	is write-only, usually an action
IUPAF_HAS_ID2	can has two IDs at the end of the name, automatically set by iupClassRegisterAttributeId2
IUPAF_CALLBACK	is a callback, not an attribute
IUPAF_NO_SAVE	can NOT be directly saved, should have at least manual processing
IUPAF_NOT_SUPPORTED	not supported in that driver
IUPAF_IHANDLENAME	is an Ihandle* name, associated with IupSetHandle
IUPAF_IHANDLE	is an Ihandle*

Function Documentation

Iclass* iupClassNew (Iclass * *ic_parent*)

Allocates memory for the Iclass structure and initializes the attribute handling functions table.
If parent is specified then a new instance of the parent class is created and set as the actual parent class.

void iupClassRelease (Iclass * *ic*)

Release the memory allocated by the class. Calls the Iclass::Release method.
Called from iupRegisterFinish.

int iupClassMatch (Iclass * *ic*,
const char * *classname*)

Check if the class name match the given name.
Parent classes are also checked.

void iupClassRegisterAttribute (Iclass * *ic*,
const char * *name*,
IattribGetFunc *get*,
IattribSetFunc *set*,
const char * *default_value*,
const char * *system_default*,
int *flags*)

Register attribute handling functions, defaults and flags. get, set and default_value can be NULL. default_value should point to a constant string, it will not be duplicated internally.
Notice that when an attribute is not defined then default_value=NULL, is inheritable can has a default value and is a string.
Since there is only one attribute function table per class tree, if you register the same attribute in a child class, then it will replace the parent registration.
If an attribute is not inheritable or not a string then it MUST be registered. Internal attributes (starting with "_IUP") can never be registered.

void iupClassRegisterAttributeId	(	Iclass *	*ic*,	
		const char *	*name*,	
		IattribGetIdFunc	*get*,	
		IattribSetIdFunc	*set*,	
		int	*flags*	
	)			

Same as iupClassRegisterAttribute for attributes with Ids.

void iupClassRegisterAttributeId2	(	Iclass *	*ic*,	
		const char *	*name*,	
		IattribGetId2Func	*get*,	
		IattribSetId2Func	*set*,	
		int	*flags*	
	)			

Same as iupClassRegisterAttribute for attributes with two Ids.

void iupClassRegisterGetAttribute	(	Iclass *	*ic*,	
		const char *	*name*,	
		IattribGetFunc *	*get*,	
		IattribSetFunc *	*set*,	
		const char **	*default_value*,	
		const char **	*system_default*,	
		int *	*flags*	
	)			

Returns the attribute handling functions, defaults and flags.

void iupClassRegisterReplaceAttribFunc	(	Iclass *	*ic*,	
		const char *	*name*,	
		IattribGetFunc	*_get*,	
		IattribSetFunc	*_set*	
	)			

Replaces the attribute handling functions of an already registered attribute.

void iupClassRegisterReplaceAttribDef	(	Iclass *	*ic*,	
		const char *	*name*,	
		const char *	*_default_value*,	
		const char *	*_system_default*	
	)			

Replaces the attribute handling default of an already registered attribute.

void iupClassRegisterReplaceAttribFlags	(	Iclass *	*ic*,	
		const char *	*name*,	
		int	*_flags*	
	)			

Replaces the attribute handling functions of an already registered attribute.

void iupClassRegisterCallback	(	Iclass *	*ic*,	
		const char *	*name*,	
		const char *	*format*	
	)			

Register the parameters of a callback.
Format follows the iupcbs::h header definitions.
Notice that these definitions are similar to the class registration but have several differences and conflicts, for backward compatibility reasons.
It can have none, one or more of the following.

- "c" = (unsigned char) - byte
- "i" = (int) - integer
- "I" = (int*) - array of integers or pointer to integer
- "f" = (float) - real
- "d" = (double) - real
- "s" = (char*) - string
- "V" = (void*) - generic pointer
- "C" = (struct _cdCanvas*) - cdCanvas* structure, used along with the CD library
- "n" = (IhandIe*) - element handle The default return value for all callbacks is "i" (int), but a different return value can be specified using one of the above parameters, after all parameters using "=" to separate it from them.

char* iupClassCallbackGetFormat	(	Iclass *	*ic*,	
		const char *	*name*	
	)			

Returns the format of the parameters of a registered callback. If NULL then the default callback definition is assumed.

Functions

Class Object Functions

[Ihandle Class]

Collaboration diagram for Class Object Functions:

Functions

int	iupClassObjectCreate (Ihandle *ih, void **params)
int	iupClassObjectMap (Ihandle *ih)
void	iupClassObjectUnMap (Ihandle *ih)
void	iupClassObjectDestroy (Ihandle *ih)
Ihandle *	iupClassObjectGetInnerContainer (Ihandle *ih)
void *	iupClassObjectGetInnerNativeContainerHandle (Ihandle *ih, Ihandle *child)
void	iupClassObjectChildAdded (Ihandle *ih, Ihandle *child)
void	iupClassObjectChildRemoved (Ihandle *ih, Ihandle *child, int pos)
void	iupClassObjectLayoutUpdate (Ihandle *ih)
void	iupClassObjectComputeNaturalSize (Ihandle *ih, int *w, int *h, int *children_expand)
void	iupClassObjectSetChildrenCurrentSize (Ihandle *ih, int shrink)
void	iupClassObjectSetChildrenPosition (Ihandle *ih, int x, int y)
int	iupClassObjectDlgPopup (Ihandle *ih, int x, int y)

Detailed Description

Stubs for the class methods. They implement inheritance and check if method is NULL.

See iup_class.h

Function Documentation

int iupClassObjectCreate (Ihandle * *ih*,
void ** *params*)

Calls Iclass::Create method.

int iupClassObjectMap (Ihandle * *ih*)

Calls Iclass::Map method.

void iupClassObjectUnMap (Ihandle * *ih*)

Calls Iclass::UnMap method.

void iupClassObjectDestroy (Ihandle * *ih*)

Calls Iclass::Destroy method.

Ihandle* iupClassObjectGetInnerContainer (Ihandle * *ih*)

Calls Iclass::GetInnerContainer method. The parent class is ignored. If necessary the child class must handle the parent class internally.

void* iupClassObjectGetInnerNativeContainerHandle (Ihandle * *ih*,
Ihandle * *child*)

Calls Iclass::GetInnerNativeContainerHandle method. Returns ih->handle if there is no inner parent. The parent class is ignored. If necessary the child class must handle the parent class internally.

void iupClassObjectChildAdded (Ihandle * *ih*,
Ihandle * *child*)

Calls Iclass::ChildAdded method.

void iupClassObjectChildRemoved (Ihandle * *ih*,
Ihandle * *child*,
int *pos*)

Calls Iclass::ChildRemoved method.

void iupClassObjectLayoutUpdate (Ihandle * *ih*)

Calls Iclass::LayoutUpdate method.

void iupClassObjectComputeNaturalSize (Ihandle * *ih*,

	int *	*w*,	
	int *	*h*,	
	int *	*children_expand*	
)			

Calls Iclass::ComputeNaturalSize method.

void iupClassObjectSetChildrenCurrentSize	(	Ihandle *	*ih*,	
		int	*shrink*	
	)			

Calls Iclass::SetChildrenCurrentSize method.

void iupClassObjectSetChildrenPosition	(	Ihandle *	*ih*,	
		int	*x*,	
		int	*y*	
	)			

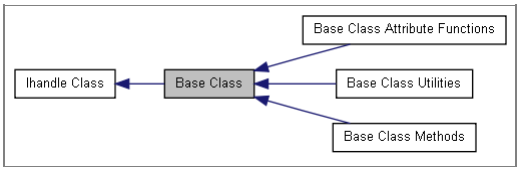
Calls Iclass::SetChildrenPosition method.

int iupClassObjectDlgPopup	(	Ihandle *	*ih*,	
		int	*x*,	
		int	*y*	
	)			

Calls Iclass::DlgPopup method.

Base Class
[Ihandle Class]

Collaboration diagram for Base Class:



Modules

	Base Class Methods
	Base Class Attribute Functions
	Base Class Utilities

Functions

void	iupBaseRegisterCommonAttrib (Iclass *ic)
void	iupBaseRegisterVisualAttrib (Iclass *ic)
void	iupBaseRegisterCommonCallbacks (Iclass *ic)
void	iupBaseContainerUpdateExpand (Ihandle *ih)
void	iupBaseComputeNaturalSize (Ihandle *ih)
void	iupBaseSetCurrentSize (Ihandle *ih, int w, int h, int shrink)
void	iupBaseSetPosition (Ihandle *ih, int x, int y)

Detailed Description

See iup_classbase.h

Function Documentation

void iupBaseRegisterCommonAttrib	(	Iclass *	*ic*	)	

Register all common base attributes:
WID
SIZE, RASTERSIZE, POSITION
FONT (and derived)

All controls that are positioned inside a dialog must register all common base attributes.

void iupBaseRegisterVisualAttrib	(	Iclass *	*ic*	)	

Register all visual base attributes:
VISIBLE, ACTIVE
ZORDER, X, Y
TIP (and derived)

All controls that are positioned inside a dialog must register all visual base attributes.

void iupBaseRegisterCommonCallbacks	(	Iclass *	*ic*	)	

Register all common callbacks:
MAP_CB, UNMAP_CB, GETFOCUS_CB, KILLFOCUS_CB, ENTERWINDOW_CB, LEAVEWINDOW_CB, K_ANY, HELP_CB.

void iupBaseContainerUpdateExpand	(	Ihandle *	*ih*	)	

Updates the expand member of the IUP object from the EXPAND attribute. Should be called in the beginning of the ComputeNaturalSize for a container.

void iupBaseComputeNaturalSize	(	Ihandle *	*ih*	)	

Initializes the natural size using the user size, then if a container then update the "expand" member from the EXPAND attribute, then call iupClassObjectComputeNaturalSize for containers if they have children or call iupClassObjectComputeNaturalSize for non-containers if user size is not defined. Must be called for each children in the container.
First call is in iupLayoutCompute.

void iupBaseSetCurrentSize	(	Ihandle *	*ih*,
		int	*w*,
		int	*h*,
		int	*shrink*
	)		

Update the current size from the available size, the natural size, expand and shrink. Call iupClassObjectSetChildrenCurrentSize for containers if they have children. Must be called for each children in the container.
First call is in iupLayoutCompute.

void iupBaseSetPosition	(	Ihandle *	*ih*,
		int	*x*,
		int	*y*
	)		

Set the current position and update children position for containers. Call iupClassObjectSetChildrenPosition for containers if they have children. Must be called for each children in the container.
First call is in iupLayoutCompute.

Base Class Methods

[Base Class]

Collaboration diagram for Base Class Methods:

Functions

void	iupdrvBaseLayoutUpdateMethod (Ihandle *ih)
void	iupdrvBaseUnMapMethod (Ihandle *ih)
int	iupBaseTypeVoidMapMethod (Ihandle *ih)

Detailed Description

See iup_classbase.h

Function Documentation

void iupdrvBaseLayoutUpdateMethod	(	Ihandle *	*ih*	)	

Driver dependent Iclass::LayoutUpdate method.

void iupdrvBaseUnMapMethod	(	Ihandle *	*ih*	)	

Driver dependent Iclass::UnMap method.

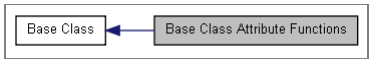
int iupBaseTypeVoidMapMethod	(	Ihandle *	*ih*	)	

Native type void Iclass::Map method.

Base Class Attribute Functions

[Base Class]

Collaboration diagram for Base Class Attribute Functions:



Functions

char *	**iupBaseGetWidAttrib** (Ihandle *ih)
int	**iupBaseSetNameAttrib** (Ihandle *ih, const char *value)
int	**iupBaseSetRasterSizeAttrib** (Ihandle *ih, const char *value)

int	**iupBaseSetSizeAttrib** (Ihandle *ih, const char *value)
char *	**iupBaseGetSizeAttrib** (Ihandle *ih)
char *	**iupBaseGetCurrentSizeAttrib** (Ihandle *ih)
char *	**iupBaseGetRasterSizeAttrib** (Ihandle *ih)
char *	**iupBaseGetClientOffsetAttrib** (Ihandle *ih)
int	**iupBaseSetMaxSizeAttrib** (Ihandle *ih, const char *value)
int	**iupBaseSetMinSizeAttrib** (Ihandle *ih, const char *value)
char *	**iupBaseGetVisibleAttrib** (Ihandle *ih)
int	**iupBaseSetVisibleAttrib** (Ihandle *ih, const char *value)
char *	**iupBaseGetActiveAttrib** (Ihandle *ih)
int	**iupBaseSetActiveAttrib** (Ihandle *ih, const char *value)
int	**iupdrvBaseSetZorderAttrib** (Ihandle *ih, const char *value)
int	**iupdrvBaseSetTipAttrib** (Ihandle *ih, const char *value)
int	**iupdrvBaseSetTipVisibleAttrib** (Ihandle *ih, const char *value)
char *	**iupdrvBaseGetTipVisibleAttrib** (Ihandle *ih)
int	**iupdrvBaseSetBgColorAttrib** (Ihandle *ih, const char *value)
int	**iupdrvBaseSetFgColorAttrib** (Ihandle *ih, const char *value)
char *	**iupBaseNativeParentGetBgColorAttrib** (Ihandle *ih)
char *	**iupBaseContainerGetExpandAttrib** (Ihandle *ih)
int	**iupdrvBaseSetCursorAttrib** (Ihandle *ih, const char *value)
void	**iupdrvRegisterDragDropAttrib** (Iclass *ic)
int	**iupBaseNoSaveCheck** (Ihandle *ih, const char *name)

Detailed Description

Used by the controls for iupClassRegisterAttribute.

See iup_classbase.h

Base Class Utilities

[Base Class]

Collaboration diagram for Base Class Utilities:

Defines

#define	**iupMAX**(_a, _b) ((_a)>(_b)?(_a):(_b))
#define	**iupROUND**(_x) ((int)((_x)>0? (_x)+0.5: (_x)-0.5))
#define	**iupCOLOR8TO16**(_x) ((unsigned short)(_x*257))
#define	**iupCOLOR16TO8**(_x) ((unsigned char)(_x/257))
#define	**iupBYTECROP**(_x) ((unsigned char)((_x)<0?0:((_x)>255)?255:(_x)))
#define	**IUP_ALIGN_ABOTTOM** IUP_ALIGN_ARIGHT
#define	**IUP_ALIGN_ATOP** IUP_ALIGN_ALEFT

Enumerations

enum	{ **IUP_ALIGN_ALEFT**, **IUP_ALIGN_ACENTER**, **IUP_ALIGN_ARIGHT** }
enum	{ **IUP_SB_NONE**, **IUP_SB_HORIZ**, **IUP_SB_VERT** }

Functions

int	**iupRound** (double x)
int	**iupBaseGetScrollbar** (Ihandle *ih)
char *	**iupBaseNativeParentGetBgColor** (Ihandle *ih)
void	**iupBaseCallValueChangedCb** (Ihandle *ih)

Detailed Description

See iup_classbase.h

Class Registration

[Control SDK]

Collaboration diagram for Class Registration:

Functions

Iclass *	iupRegisterFindClass (const char *name)
void	iupRegisterClass (Iclass *ic)

Detailed Description

All controls are registered so the creation using IupCreate can work.

See iup_register.h

Function Documentation

Iclass* iupRegisterFindClass (const char * *name*)

Returns a class instance from a class name. The class name must be previously registered using iupRegisterClass.

void iupRegisterClass (Iclass * *ic*)

Register a class.

Attribute Environment
[Control SDK]

Collaboration diagram for Attribute Environment:

Defines

#define	iupATTRIB_ISINTERNAL(_name) ((_name[0] == '_' && _name[1] == 'I' && _name[2] == 'U' && _name[3] == 'P')? 1: 0)

Functions

int	iupAttribIsNotString (Ihandle *ih, const char *name)
int	iupAttribIsIhandle (Ihandle *ih, const char *name)
void	iupAttribSet (Ihandle *ih, const char *name, const char *value)
void	iupAttribSetStr (Ihandle *ih, const char *name, const char *value)
void	iupAttribSetStrf (Ihandle *ih, const char *name, const char *format,...)
void	iupAttribSetInt (Ihandle *ih, const char *name, int num)
void	iupAttribSetId (Ihandle *ih, const char *name, int id, const char *value)
void	iupAttribSetStrId (Ihandle *ih, const char *name, int id, const char *value)
void	iupAttribSetId2 (Ihandle *ih, const char *name, int lin, int col, const char *value)
void	iupAttribSetStrId2 (Ihandle *ih, const char *name, int lin, int col, const char *value)
void	iupAttribSetIntId (Ihandle *ih, const char *name, int id, int num)
void	iupAttribSetIntId2 (Ihandle *ih, const char *name, int lin, int col, int num)
void	iupAttribSetFloat (Ihandle *ih, const char *name, float num)
void	iupAttribSetFloatId (Ihandle *ih, const char *name, int id, float num)
void	iupAttribSetFloatId2 (Ihandle *ih, const char *name, int lin, int col, float num)
void	iupAttribSetDouble (Ihandle *ih, const char *name, double num)
void	iupAttribSetDoubleId (Ihandle *ih, const char *name, int id, double num)
void	iupAttribSetDoubleId2 (Ihandle *ih, const char *name, int lin, int col, double num)
char *	iupAttribGet (Ihandle *ih, const char *name)
char *	iupAttribGetStr (Ihandle *ih, const char *name)
int	iupAttribGetInt (Ihandle *ih, const char *name)
int	iupAttribGetBoolean (Ihandle *ih, const char *name)
float	iupAttribGetFloat (Ihandle *ih, const char *name)
double	iupAttribGetDouble (Ihandle *ih, const char *name)
char *	iupAttribGetId (Ihandle *ih, const char *name, int id)
int	iupAttribGetIntId (Ihandle *ih, const char *name, int id)
int	iupAttribGetBooleanId (Ihandle *ih, const char *name, int id)
float	iupAttribGetFloatId (Ihandle *ih, const char *name, int id)
char *	iupAttribGetId2 (Ihandle *ih, const char *name, int lin, int col)
int	iupAttribGetIntId2 (Ihandle *ih, const char *name, int lin, int col)
int	iupAttribGetBooleanId2 (Ihandle *ih, const char *name, int lin, int col)
float	iupAttribGetFloatId2 (Ihandle *ih, const char *name, int lin, int col)
char *	iupAttribGetInherit (Ihandle *ih, const char *name)

char *	iupAttribGetInheritNativeParent (Ihandle *ih, const char *name)
char *	iupAttribGetLocal (Ihandle *ih, const char *name)
void	iupAttribSetHandleName (Ihandle *ih)
char *	iupAttribGetHandleName (Ihandle *ih)
void	iupAttribSetClassObject (Ihandle *ih, const char *name, const char *value)

Detailed Description

When attributes are not stored at the control they are stored in a hash table (see Hash Table).

As a general rule use:

- IupGetAttribute, IupSetAttribute, ... : when care about control implementation, hash table, inheritance and default value
- iupAttribGetStr,Int,Float: when care about inheritance, hash table and default value
- iupAttribGet,... : ONLY access the hash table These different functions have very different performances and results. So use them wiselly.

See iup_attrib.h

Define Documentation

#define iupATTRIB_ISINTERNAL	(	_name	)	((_name[0] == '_' && _name[1] == 'I' && _name[2] == 'U' && _name[3] == 'P')? 1: 0)

Returns true if the attribute name if in the internal format "_IUP...".

Function Documentation

int iupAttribIsNotString	(	Ihandle *	*ih*,	
		const char *	*name*	
	)			

Returns true if the attribute name is a known pointer.

int iupAttribIsIhandle	(	Ihandle *	*ih*,	
		const char *	*name*	
	)			

Returns true if the attribute name is a known Ihandle*.

void iupAttribSet	(	Ihandle *	*ih*,	
		const char *	*name*,	
		const char *	*value*	
	)			

Sets the attribute only in the hash table as a pointer.
Only generic pointers and constant strings can be set as value.
It ignores children.

void iupAttribSetStr	(	Ihandle *	*ih*,	
		const char *	*name*,	
		const char *	*value*	
	)			

Sets the attribute only in the hash table as a string.
The string is internally duplicated.
It ignores children.

void iupAttribSetStrf	(	Ihandle *	*ih*,	
		const char *	*name*,	
		const char *	*format*,	
			...	
	)			

Sets the attribute only in the hash table as a string.
Use same format as sprintf.
It ignores children.
This is not supposed to be used for very large strings, just for combinations of numeric data or constant strings.

void iupAttribSetInt	(	Ihandle *	*ih*,	
		const char *	*name*,	
		int	*num*	
	)			

Sets an integer attribute only in the hash table.
It will be stored as a string.
It ignores children.

void iupAttribSetId	(	Ihandle *	*ih*,	
		const char *	*name*,	
		int	*id*,	
		const char *	*value*	
	)			

Same as iupAttribSet with an id.

void iupAttribSetStrId	(	Ihandle *	*ih*,
		const char *	*name*,
		int	*id*,
		const char *	*value*
	)		

Same as iupAttribSetStr with an id.

void iupAttribSetId2	(	Ihandle *	*ih*,
		const char *	*name*,
		int	*lin*,
		int	*col*,
		const char *	*value*
	)		

Same as iupAttribSet with lin and col.

void iupAttribSetStrId2	(	Ihandle *	*ih*,
		const char *	*name*,
		int	*lin*,
		int	*col*,
		const char *	*value*
	)		

Same as iupAttribSetStr with lin and col.

void iupAttribSetIntId	(	Ihandle *	*ih*,
		const char *	*name*,
		int	*id*,
		int	*num*
	)		

Same as iupAttribSetInt with an id.

void iupAttribSetIntId2	(	Ihandle *	*ih*,
		const char *	*name*,
		int	*lin*,
		int	*col*,
		int	*num*
	)		

Same as iupAttribSetInt with lin and col.

void iupAttribSetFloat	(	Ihandle *	*ih*,
		const char *	*name*,
		float	*num*
	)		

Sets an floating point attribute only in the hash table.
It will be stored as a string.
It ignores children.

void iupAttribSetFloatId	(	Ihandle *	*ih*,
		const char *	*name*,
		int	*id*,
		float	*num*
	)		

Same as iupAttribSetFloat with an id.

void iupAttribSetFloatId2	(	Ihandle *	*ih*,
		const char *	*name*,
		int	*lin*,
		int	*col*,
		float	*num*
	)		

Same as iupAttribSetFloat with lin and col.

void iupAttribSetDouble	(	Ihandle *	*ih*,
		const char *	*name*,
		double	*num*
	)		

Sets an floating point attribute only in the hash table.
It will be stored as a string.
It ignores children.

void iupAttribSetDoubleId	(	Ihandle *	*ih*,

		const char *	*name*,	
		int	*id*,	
		double	*num*	
	)			

Same as iupAttribSetDouble with an id.

void iupAttribSetDoubleId2	(	Ihandle *	*ih*,	
		const char *	*name*,	
		int	*lin*,	
		int	*col*,	
		double	*num*	
	)			

Same as iupAttribSetDouble with lin and col.

char* iupAttribGet	(	Ihandle *	*ih*,	
		const char *	*name*	
	)			

Returns the attribute from the hash table only.
NO inheritance, NO control implementation, NO defalt value here.

char* iupAttribGetStr	(	Ihandle *	*ih*,	
		const char *	*name*	
	)			

Returns the attribute from the hash table as a string, but if not defined then checks in its parent tree if allowed by the control implementation, if still not defined then returns the registered default value if any. NO control implementation, only checks inheritance and default value from it.

int iupAttribGetInt	(	Ihandle *	*ih*,	
		const char *	*name*	
	)			

Same as iupAttribGetStr but returns an integer number. Checks also for boolean values.

int iupAttribGetBoolean	(	Ihandle *	*ih*,	
		const char *	*name*	
	)			

Same as iupAttribGetStr but checks for boolean values. Use iupStrBoolean.

float iupAttribGetFloat	(	Ihandle *	*ih*,	
		const char *	*name*	
	)			

Same as iupAttribGetStr but returns an floating point number.

double iupAttribGetDouble	(	Ihandle *	*ih*,	
		const char *	*name*	
	)			

Same as iupAttribGetStr but returns an floating point number.

char* iupAttribGetId	(	Ihandle *	*ih*,	
		const char *	*name*,	
		int	*id*	
	)			

Same as iupAttribGet but with an id.

int iupAttribGetIntId	(	Ihandle *	*ih*,	
		const char *	*name*,	
		int	*id*	
	)			

Same as iupAttribGetInt but with an id.

int iupAttribGetBooleanId	(	Ihandle *	*ih*,	
		const char *	*name*,	
		int	*id*	
	)			

Same as iupAttribGetBoolean but with an id.

float iupAttribGetFloatId	(	Ihandle *	*ih*,	
		const char *	*name*,	
		int	*id*	
	)			

Same as iupAttribGetFloat but with an id.

char* iupAttribGetId2	(	Ihandle *	*ih*,	

		const char *	*name*,
		int	*lin*,
		int	*col*
	)		

Same as iupAttribGet but with an lin and col.

int iupAttribGetIntId2	(	Ihandle *	*ih*,
		const char *	*name*,
		int	*lin*,
		int	*col*
	)		

Same as iupAttribGetInt but with lin and col.

int iupAttribGetBooleanId2	(	Ihandle *	*ih*,
		const char *	*name*,
		int	*lin*,
		int	*col*
	)		

Same as iupAttribGetBoolean but with lin and col.

float iupAttribGetFloatId2	(	Ihandle *	*ih*,
		const char *	*name*,
		int	*lin*,
		int	*col*
	)		

Same as iupAttribGetFloat but with lin and col.

char* iupAttribGetInherit	(	Ihandle *	*ih*,
		const char *	*name*
	)		

Returns the attribute from the hash table only, but if not defined then checks in its parent tree.
NO control implementation, NO defalt value here.
Used for EXPAND and internal attributes inside a dialog.

char* iupAttribGetInheritNativeParent	(	Ihandle *	*ih*,
		const char *	*name*
	)		

Returns the attribute from the hash table of a native parent. Don't check for default values. Don't check at the element. Used for BGCOLOR and BACKGROUND attributes.

char* iupAttribGetLocal	(	Ihandle *	*ih*,
		const char *	*name*
	)		

Returns the attribute from the hash table as a string, but if not defined then checks in the control implementation, if still not defined then returns the registered default value if any.
NO inheritance here. Used only in the IupLayoutDialog.

void iupAttribSetHandleName	(	Ihandle *	*ih*	)

Set an internal name to a handle.

char* iupAttribGetHandleName	(	Ihandle *	*ih*	)

Returns the internal name if set.

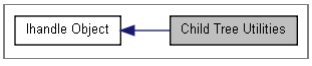
void iupAttribSetClassObject	(	Ihandle *	*ih*,
		const char *	*name*,
		const char *	*value*
	)		

Sets the attribute only at the element.
If set method is not defined will not be set, neither will be stored in the hash table.
Only generic pointers and constant strings can be set as value.
It ignores children.

Child Tree Utilities

[Ihandle Object]

Collaboration diagram for Child Tree Utilities:



Functions

Ihandle *	iupChildTreeGetNativeParent (Ihandle *ih)
InativeHandle *	iupChildTreeGetNativeParentHandle (Ihandle *ih)
void	iupChildTreeAppend (Ihandle *parent, Ihandle *child)
int	iupChildTreeIsChild (Ihandle *ih, Ihandle *child)
Ihandle *	iupChildTreeGetPrevBrother (Ihandle *ih)

Detailed Description

Some native containers have an internal native child that will be the actual container for the children. This native container is returned by iupClassObjectGetInnerNativeContainerHandle and it is used in iupChildTreeGetNativeParentHandle.

Some native elements need an extra parent, the ih->handle points to the main element itself, NOT to the extra parent. This extra parent is stored as "_IUP_EXTRAPARENT". In this case the native parent of ih->handle is the extra parent, and the extra parent is added as child to the element actual native parent.

See iup_childtree.h

Function Documentation

Ihandle* iupChildTreeGetNativeParent (Ihandle * *ih*)

Returns the native parent. It simply excludes containers that are from IUP_TYPEVOID classes.

InativeHandle* iupChildTreeGetNativeParentHandle (Ihandle * *ih*)

Returns the native parent handle. Uses iupChildTreeGetNativeParent and iupClassObjectGetInnerNativeContainerHandle.

void iupChildTreeAppend (Ihandle * *parent*,
Ihandle * *child*
)

Adds the child directly to the parent tree.

int iupChildTreeIsChild (Ihandle * *ih*,
Ihandle * *child*
)

Checks if the child belongs to the parent tree.

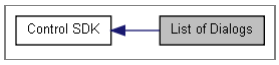
Ihandle* iupChildTreeGetPrevBrother (Ihandle * *ih*)

Returns the previous brother if any.

Functions

List of Dialogs
[Control SDK]

Collaboration diagram for List of Dialogs:



Functions

void	iupDlgListAdd (Ihandle *ih)
void	iupDlgListRemove (Ihandle *ih)
int	iupDlgListCount (void)
Ihandle *	iupDlgListFirst (void)
Ihandle *	iupDlgListNext (void)
void	iupDlgListVisibleInc (void)
void	iupDlgListVisibleDec (void)
int	iupDlgListVisibleCount (void)

Detailed Description

See iup_dlglist.h

Function Documentation

void iupDlgListAdd (Ihandle * *ih*)

Adds a dialog to the list. Used only in IupDialog.

void iupDlgListRemove (Ihandle * *ih*)

Removes a dialog from the list. Used only in IupDestroy.

int iupDlgListCount (void)

Returns the number of dialogs.

Ihandle* iupDlgListFirst (void)

Starts a loop for all the created dialogs.

Ihandle* iupDlgListNext (void)

Retrieve the next dialog on the list. Must call iupDlgListFirst first.

void iupDlgListVisibleInc (void)

Increments the number of visible dialogs.

void iupDlgListVisibleDec (void)

Decrements the number of visible dialogs.

int iupDlgListVisibleCount (void)

Returns the number of visible dialogs.

Keyboard Focus
[Control SDK]

Collaboration diagram for Keyboard Focus:

Functions

int	iupFocusCanAccept (Ihandle *ih)
void	iupCallGetFocusCb (Ihandle *ih)
void	iupCallKillFocusCb (Ihandle *ih)
Ihandle *	iupFocusNextInteractive (Ihandle *ih)

Detailed Description

See iup_focus.h

Function Documentation

int iupFocusCanAccept (Ihandle * *ih*)

Utility to check if a control can have the keyboard input focus. To receive the focus must be interactive, has CANFOCUS=YES, is mapped, is visible and is active.

void iupCallGetFocusCb (Ihandle * *ih*)

Call GETFOCUS_CB and FOCUS_CB.

void iupCallKillFocusCb (Ihandle * *ih*)

Call KILLFOCUS_CB and FOCUS_CB.

Ihandle* iupFocusNextInteractive (Ihandle * *ih*)

Returns the next interactive brother. Independs if it can receive the focus.

Key Coding and Key Callbacks
[Control SDK]

Collaboration diagram for Key Coding and Key Callbacks:

Functions

char *	iupKeyCodeToName (int code)
void	iupKeyForEach (void(*func)(const char *name, int code, void *user_data), void *user_data)
int	iupKeyCallKeyCb (Ihandle *ih, int c)
int	iupKeyCallKeyPressCb (Ihandle *ih, int code, int press)
int	iupKeyProcessNavigation (Ihandle *ih, int code, int shift)
int	iupKeyProcessMnemonic (Ihandle *ih, int code)
void	iupKeySetMnemonic (Ihandle *ih, int code, int pos)

Detailed Description

See iup_key.h

Function Documentation

char* iupKeyCodeToName	(	int	*code*	)		

Returns the key name from its code. Returns NULL if code not found.

void iupKeyForEach	(	void(*)(const char *name, int code, void *user_data)	*func*,	
		void *	*user_data*	
	)			

Calls a function for each defined key.
Used only by the IupLua binding.

int iupKeyCallKeyCb	(	Ihandle *	*ih*,	
		int	*c*	
	)			

Calls the K_ANY or K_* callbacks. Should be called when a keyboard event occoured.

int iupKeyCallKeyPressCb	(	Ihandle *	*ih*,	
		int	*code*,	
		int	*press*	
	)			

Calls the KEYPRESS_CB callback. Should be called when a keyboard event occoured.

int iupKeyProcessNavigation	(	Ihandle *	*ih*,	
		int	*code*,	
		int	*shift*	
	)			

Process Tab, DEFAULTENTER and DEFAULTESC in key press events.

int iupKeyProcessMnemonic	(	Ihandle *	*ih*,	
		int	*code*	
	)			

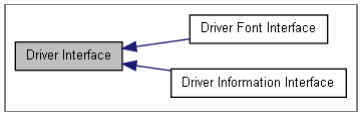
Process mnemonics (Used only in Windows and Motif).

void iupKeySetMnemonic	(	Ihandle *	*ih*,	
		int	*code*,	
		int	*pos*	
	)			

Set a mnemonic (Used only in Windows and Motif).

Driver Interface

Collaboration diagram for Driver Interface:



Modules

	Driver Font Interface
	Driver Information Interface

Functions

int	iupdrvSetGlobal (const char *name, const char *value)
char *	iupdrvGetGlobal (const char *name)
void	iupdrvSetIdleFunction (Icallback func)
void	iupdrvScreenToClient (Ihandle *ih, int *x, int *y)
void	iupdrvClientToScreen (Ihandle *ih, int *x, int *y)
int	iupdrvIsVisible (Ihandle *ih)
int	iupdrvIsActive (Ihandle *ih)
void	iupdrvSetFocus (Ihandle *ih)
void	iupdrvSetVisible (Ihandle *ih, int enable)
void	iupdrvSetActive (Ihandle *ih, int enable)
void	iupdrvPostRedraw (Ihandle *ih)
void	iupdrvRedrawNow (Ihandle *ih)
void	iupdrvReparent (Ihandle *ih)
void	iupdrvDrawFocusRect (Ihandle *ih, void *gc, int x, int y, int w, int h)

int	iupdrvGetScrollbarSize (void)
void	iupdrvActivate (Ihandle *ih)
int	iupdrvMenuGetMenuBarSize (Ihandle *ih)
void	iupdrvSendKey (int key, int press)
void	iupdrvSendMouse (int x, int y, int bt, int status)
void	iupdrvWarpPointer (int x, int y)
void	iupdrvKeyEncode (int key, unsigned int *keyval, unsigned int *state)
void	iupdrvSleep (int time)

Detailed Description

Each driver must export the symbols defined here.

See iup_drv.h

Function Documentation

int iupdrvSetGlobal	(	const char *	*name*,	
		const char *	*value*	
	)			

Sets a global environment attribute. Called from IupSetGlobal and IupStoreGlobal. Must return 1 is process the attribute, or 0 is not.

char* iupdrvGetGlobal	(	const char *	*name*	)

Returns a global environment attribute. Called from IupGetGlobal.

void iupdrvSetIdleFunction	(	Icallback	*func*	)

Changes the idle callback. Called from IupSetFunction.

void iupdrvScreenToClient	(	Ihandle *	*ih*,	
		int *	*x*,	
		int *	*y*	
	)			

Convert the coordinates from screen relative to client area.

void iupdrvClientToScreen	(	Ihandle *	*ih*,	
		int *	*x*,	
		int *	*y*	
	)			

Convert the coordinates from relative client area to screen.

int iupdrvIsVisible	(	Ihandle *	*ih*	)

Returns true if the element is visible.

int iupdrvIsActive	(	Ihandle *	*ih*	)

Returns true if the element is active.

void iupdrvSetFocus	(	Ihandle *	*ih*	)

Actually changes the focus to the given element.

void iupdrvSetVisible	(	Ihandle *	*ih*,	
		int	*enable*	
	)			

Changes the visible state of an element. Not used for dialogs.

void iupdrvSetActive	(	Ihandle *	*ih*,	
		int	*enable*	
	)			

Changes the active state of an element.

void iupdrvPostRedraw	(	Ihandle *	*ih*	)

Post a redraw of a control.

void iupdrvRedrawNow	(	Ihandle *	*ih*	)

Force a redraw of a control.

void iupdrvReparent	(	Ihandle *	*ih*	)

Reparent the native control.

void iupdrvDrawFocusRect	(	Ihandle *	*ih*,	
		void *	*gc*,	
		int	*x*,	
		int	*y*,	

		int	*w*,	
		int	*h*	
	)			

Draws a focus rectangle gc is:

- HDC in Win32
- GC in Motif
- unused in GTK2
- cairo_t in GTK3 When using CD, use the "GC" CD canvas attribute.

int iupdrvGetScrollbarSize	(	void	)	

Size of the scrollbar.

void iupdrvActivate	(	Ihandle *	*ih*	)

Activates a button or toggle.

int iupdrvMenuGetMenuBarSize	(	Ihandle *	*ih*	)

Returns the height of a menu bar.

void iupdrvSendKey	(	int	*key*,	
		int	*press*	
	)			

Sends a global keyboard message.

void iupdrvSendMouse	(	int	*x*,	
		int	*y*,	
		int	*bt*,	
		int	*status*	
	)			

Sends a global mouse message. status: 2=double pressed, 1=pressed, 0=released, -1=move

void iupdrvWarpPointer	(	int	*x*,	
		int	*y*	
	)			

Moves the cursor on screen.

void iupdrvKeyEncode	(	int	*key*,	
		unsigned int *	*keyval*,	
		unsigned int *	*state*	
	)			

Translates an IUP key definition into a system definition.

void iupdrvSleep	(	int	*time*	)

Suspends execution for the specified number of milliseconds.

Functions

Driver Font Interface

[Driver Interface]

Collaboration diagram for Driver Font Interface:

Functions

void	iupdrvFontGetCharSize (Ihandle *ih, int *charwidth, int *charheight)
int	iupdrvFontGetStringWidth (Ihandle *ih, const char *str)
void	iupdrvFontGetMultiLineStringSize (Ihandle *ih, const char *str, int *w, int *h)
char *	iupdrvGetSystemFont (void)
int	iupdrvSetStandardFontAttrib (Ihandle *ih, const char *value)
char *	iupGetFontAttrib (Ihandle *ih)
int	iupSetFontAttrib (Ihandle *ih, const char *value)
int	iupGetFontInfo (const char *standardfont, char *typeface, int *size, int *is_bold, int *is_italic, int *is_underline, int *is_strikeout)
int	iupFontParsePango (const char *value, char *typeface, int *size, int *bold, int *italic, int *underline, int *strikeout)
int	iupFontParseWin (const char *value, char *typeface, int *size, int *bold, int *italic, int *underline, int *strikeout)
int	iupFontParseX (const char *value, char *typeface, int *size, int *bold, int *italic, int *underline, int *strikeout)

Detailed Description

Each driver must export the symbols defined here.

See iup_drvfont.h

Function Documentation

void iupdrvFontGetCharSize	(	Ihandle *	*ih*,	
		int *	*charwidth*,	
		int *	*charheight*	
	)			

Retrieve the character size for the selected font. Should be used only to calculate the SIZE attribute.

int iupdrvFontGetStringWidth	(	Ihandle *	*ih*,	
		const char *	*str*	
	)			

Retrieve the string width for the selected font.

void iupdrvFontGetMultiLineStringSize	(	Ihandle *	*ih*,	
		const char *	*str*,	
		int *	*w*,	
		int *	*h*	
	)			

Retrieve the multi-lined string size for the selected font.
Width is the maximum line width.
Height is charheight*number_of_lines (this will avoid line size variations).

char* iupdrvGetSystemFont	(	void	)

Returns the System default font.

int iupdrvSetStandardFontAttrib	(	Ihandle *	*ih*,	
		const char *	*value*	
	)			

STANDARDFONT attribute set function.

char* iupGetFontAttrib	(	Ihandle *	*ih*	)

FONT attribute get function.

int iupSetFontAttrib	(	Ihandle *	*ih*,	
		const char *	*value*	
	)			

FONT attribute set function.

int iupGetFontInfo	(	const char *	*standardfont*,
		char *	*typeface*,
		int *	*size*,
		int *	*is_bold*,
		int *	*is_italic*,
		int *	*is_underline*,
		int *	*is_strikeout*
	)		

Parse the font format description. Returns a non zero value if successful.

int iupFontParsePango	(	const char *	*value*,
		char *	*typeface*,
		int *	*size*,
		int *	*bold*,
		int *	*italic*,
		int *	*underline*,
		int *	*strikeout*
	)		

Parse the Pango font format description. Returns a non zero value if successful.

int iupFontParseWin	(	const char *	*value*,
		char *	*typeface*,
		int *	*size*,
		int *	*bold*,
		int *	*italic*,
		int *	*underline*,
		int *	*strikeout*
	)		

Parse the old IUP Windows font format description. Returns a non zero value if successful.

int iupFontParseX	(	const char *	*value*,	
		char *	*typeface*,	
		int *	*size*,	
		int *	*bold*,	
		int *	*italic*,	
		int *	*underline*,	
		int *	*strikeout*	
	)			

Parse the X-Windows font format description. Returns a non zero value if successful.

Functions

Driver Information Interface

[Driver Interface]

Collaboration diagram for Driver Information Interface:

Functions

void	iupdrvGetFullSize (int *width, int *height)
void	iupdrvGetScreenSize (int *width, int *height)
void	iupdrvAddScreenOffset (int *x, int *y, int add)
int	iupdrvCheckMainScreen (int *width, int *height)
int	iupdrvGetScreenDepth (void)
float	iupdrvGetScreenDpi (void)
char *	iupdrvGetSystemVersion (void)
char *	iupdrvGetSystemName (void)
char *	iupdrvGetComputerName (void)
char *	iupdrvGetUserName (void)
void	iupdrvGetKeyState (char *key)
void	iupdrvGetCursorPos (int *x, int *y)
void *	iupdrvGetDisplay (void)
char *	iupdrvLocaleInfo (void)

Detailed Description

Each driver must export the symbols defined here. But in this case the functions are shared by different drivers in the same system.

For example, the GTK driver and the Windows driver share the same implementation of these functions when the GTK driver is compiled in Windows. The GTK driver and the Motif driver share the same implementation of these functions when the GTK driver is compiled in UNIX.

See iup_drvinfo.h

Function Documentation

void iupdrvGetFullSize	(	int *	*width*,	
		int *	*height*	
	)			

Retrieve the main desktop full size.

void iupdrvGetScreenSize	(	int *	*width*,	
		int *	*height*	
	)			

Retrieve the main desktop available size.

void iupdrvAddScreenOffset	(	int *	*x*,	
		int *	*y*,	
		int	*add*	
	)			

Adds the main desktop offset because of a taskbar/menubar positioning. Only useful in Windows. In X-Windows the position of the origin 0,0 is already adjusted to be after the taskbar/menubar.

int iupdrvCheckMainScreen	(	int *	*width*,	
		int *	*height*	
	)			

Retrieve the main desktop size when there are multiple monitors. Useful only when in GTK.

int iupdrvGetScreenDepth	(	void		)

Retrieve the default desktop bits per pixel.

float iupdrvGetScreenDpi	(	void	)

Retrieve the default desktop resolution in dpi (dots or pixels per inch).

char* iupdrvGetSystemVersion	(	void	)

Returns a string with the system version number.

char* iupdrvGetSystemName	(	void	)

Returns a string with the system name.

char* iupdrvGetComputerName	(	void	)

Returns a string with the computer name.

char* iupdrvGetUserName	(	void	)

Returns a string with the user name.

void iupdrvGetKeyState	(	char *	*key*	)

Returns the key state for Shift, Ctrl, Alt and sYs, in this order. Left and right keys are considered. Should declare "char key[5]". Values could be space (" ") or "SCAY".

void iupdrvGetCursorPos	(	int *	*x*,
		int *	*y*
	)		

Returns the current position of the mouse cursor.

void* iupdrvGetDisplay	(	void	)

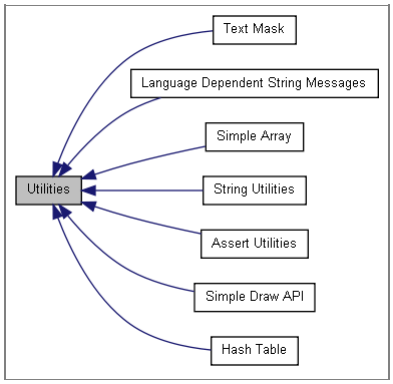
Returns the driver "Display" in UNIX and NULL in Windows. Must be implemented somewhere else.

char* iupdrvLocaleInfo	(	void	)

Returns the current locale name.

Utilities

Collaboration diagram for Utilities:



Modules

- Simple Array
- Assert Utilities
- Simple Draw API
- Text Mask
- String Utilities
- Language Dependent String Messages
- Hash Table

Assert Utilities

[Utilities]

Collaboration diagram for Assert Utilities:

Defines

#define	iupASSERT(_expr) ((_expr)? (void)0: iupAssert(#_expr, __FILE__, __LINE__, NULL))

#define	iupERROR(_msg) iupError(_msg)

Detailed Description

All functions of the main API (Iup***) calls iupASSERT to check the parameters.

The IUP main library must be recompiled with the IUP_ASSERT define to enable these checks. iupASSERT is not called inside driver dependent functions nor in each control implementation, it is used only in the functions of the main API and in some utilities.

See iup_assert.h

Define Documentation

#define iupASSERT	(	_expr	)	((_expr)? (void)0: iupAssert(#_expr, __FILE__, __LINE__, NULL))

If the expression if false, displays a message with information of the source code where the assert happen.

Parameters:

	_expr	The evaluated expression.

It is a macro that calls a function only if IUP_ASSERT is defined.

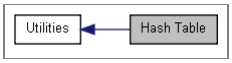
#define iupERROR	(	_msg	)	iupError(_msg)

Displays an error message. Also used by the iupASSERT.

It is a macro that calls a function only if IUP_ASSERT is defined.

Hash Table
[Utilities]

Collaboration diagram for Hash Table:



Typedefs

typedef enum _Itable_IndexTypes	Itable_IndexTypes
typedef enum _Itable_Types	Itable_Types

Enumerations

enum	_Itable_IndexTypes { IUPTABLE_POINTERINDEXED = 10, IUPTABLE_STRINGINDEXED }
enum	_Itable_Types { IUPTABLE_POINTER, IUPTABLE_STRING, IUPTABLE_FUNCPOINTER }

Functions

Itable *	iupTableCreate (Itable_IndexTypes indexType)
Itable *	iupTableCreateSized (Itable_IndexTypes indexType, unsigned int initialSizeIndex)
void	iupTableDestroy (Itable *it)
void	iupTableClear (Itable *it)
int	iupTableCount (Itable *it)
void	iupTableSet (Itable *it, const char *key, void *value, Itable_Types itemType)
void	iupTableSetFunc (Itable *it, const char *key, Ifunc func)
void *	iupTableGet (Itable *it, const char *key)
Ifunc	iupTableGetFunc (Itable *it, const char *key, void **value)
void *	iupTableGetTyped (Itable *it, const char *key, Itable_Types *itemType)
void	iupTableRemove (Itable *it, const char *key)
char *	iupTableFirst (Itable *it)
char *	iupTableNext (Itable *it)
void *	iupTableGetCurr (Itable *it)
int	iupTableGetCurrType (Itable *it)
void	iupTableSetCurr (Itable *it, void *value, Itable_Types itemType)
char *	iupTableRemoveCurr (Itable *it)

Detailed Description

The hash table can be indexed by strings or pointer address, and each value can contain strings, pointers or function pointers.

See iup_table.h

Typedef Documentation

typedef enum _Itable_IndexTypes Itable_IndexTypes

How the table key is interpreted.

typedef enum _Itable_Types Itable_Types

How the value is interpreted.

Enumeration Type Documentation

enum _Itable_IndexTypes

How the table key is interpreted.

Enumerator:

IUPTABLE_POINTERINDEXED	a pointer address is used as key.
IUPTABLE_STRINGINDEXED	a string as key

enum _Itable_Types

How the value is interpreted.

Enumerator:

IUPTABLE_POINTER	regular pointer for strings and other pointers
IUPTABLE_STRING	string duplicated internally
IUPTABLE_FUNCPOINTER	function pointer

Function Documentation

Itable* iupTableCreate (Itable_IndexTypes *indexType*)

Creates a hash table with an initial default size. This function is equivalent to iupTableCreateSized(0);

Itable* iupTableCreateSized (Itable_IndexTypes *indexType*,
unsigned int *initialSizeIndex*)

Creates a hash table with the specified initial size. Use this function if you expect the table to become very large. initialSizeIndex is an array into the (internal) list of possible hash table sizes. Currently only indexes from 0 to 8 are supported. If you specify a higher value here, the maximum allowed value will be used.

void iupTableDestroy (Itable * *it*)

Destroys the Itable. Calls iupTableClear.

void iupTableClear (Itable * *it*)

Removes all items in the table. This function does also free the memory of strings contained in the table!!!!

int iupTableCount (Itable * *it*)

Returns the number of keys stored in the table.

void iupTableSet (Itable * *it*,
const char * *key*,
void * *value*,
Itable_Types *itemType*)

Store an element in the table.

void iupTableSetFunc (Itable * *it*,
const char * *key*,
Ifunc *func*)

Store a function pointer in the table. Type is set to IUPTABLE_FUNCPOINTER.

void* iupTableGet (Itable * *it*,
const char * *key*)

Retrieves an element from the table. Returns NULL if not found.

Ifunc iupTableGetFunc (Itable * *it*,
const char * *key*,
void ** *value*)

Retrieves a function pointer from the table. If not a function or not found returns NULL. value always contains the element pointer.

void* iupTableGetTyped (Itable * *it*,

		const char *	*key*,	
		Itable_Types *	*itemType*	
	)			

Retrieves an element from the table and its type.

void iupTableRemove	(	Itable *	*it*,	
		const char *	*key*	
	)			

Removes the entry at the specified key from the hash table and frees the memory used by it if it is a string...

char* iupTableFirst	(	Itable *	*it*	)

Key iteration function. Returns a key. To iterate over all keys call iupTableFirst at the first and call iupTableNext in a loop until 0 is returned... Do NOT change the content of the hash table during iteration. During an iteration you can use context with iupTableGetCurr() to access the value of the key very fast.

char* iupTableNext	(	Itable *	*it*	)

Key iteration function. See iupTableNext.

void* iupTableGetCurr	(	Itable *	*it*	)

Returns the value at the current position.
The current context is an iterator that is filled by iupTableNext().
iupTableGetCur() is faster then iupTableGet(), so when you want to access an item stored at a key returned by iupTableNext(), use this function instead of iupTableGet().

int iupTableGetCurrType	(	Itable *	*it*	)

Returns the type at the current position.
Same as iupTableGetCurr but returns the type. Returns -1 if failed.

void iupTableSetCurr	(	Itable *	*it*,	
		void *	*value*,	
		Itable_Types	*itemType*	
	)			

Replaces the data at the current position.

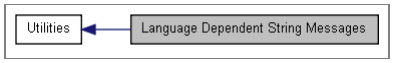
char* iupTableRemoveCurr	(	Itable *	*it*	)

Removes the current element and returns the next key. Use this function to remove an element during an iteration.

Language Dependent String Messages

[Utilities]

Collaboration diagram for Language Dependent String Messages:



String database that is dependend of the selected language.

See iup_strmessage.h

Functions

Simple Array

[Utilities]

Collaboration diagram for Simple Array:

Functions

Iarray *	iupArrayCreate (int start_max_count, int elem_size)
void	iupArrayDestroy (Iarray *iarray)
void *	iupArrayGetData (Iarray *iarray)
void *	iupArrayInc (Iarray *iarray)
void *	iupArrayAdd (Iarray *iarray, int add_count)
void *	iupArrayInsert (Iarray *iarray, int index, int insert_count)
void	iupArrayRemove (Iarray *iarray, int index, int remove_count)
int	iupArrayCount (Iarray *iarray)

Detailed Description

Expandable array using a simple pointer.

See iup_array.h

Function Documentation

```
Iarray* iupArrayCreate ( int start_max_count,
                         int elem_size
                       )
```

Creates an array with an initial room for elements, and the element size. The array count starts at 0. And the maximum number of elements starts at the given count. The maximum number of elements is increased by the start_max_count, every time it needs more memory. Data is always initialized with zeros. Must call iupArrayInc to proper allocates memory.

```
void iupArrayDestroy ( Iarray * iarray )
```

Destroys the array.

```
void* iupArrayGetData ( Iarray * iarray )
```

Returns the pointer that contains the array.

```
void* iupArrayInc ( Iarray * iarray )
```

Increments the number of elements in the array. The array count starts at 0. If the maximum number of elements is reached, the memory allocated is increased by the initial start count. Data is always initialized with zeros. Returns the pointer that contains the array.

```
void* iupArrayAdd ( Iarray * iarray,
                    int add_count
                  )
```

Increments the number of elements in the array by a given count. New space is allocated at the end of the array. If the maximum number of elements is reached, the memory allocated is increased by the given count. Data is always initialized with zeros. Returns the pointer that contains the array.

```
void* iupArrayInsert ( Iarray * iarray,
                       int index,
                       int insert_count
                     )
```

Increments the number of elements in the array by a given count and moves the data so the new space starts at index. If the maximum number of elements is reached, the memory allocated is increased by the given count. Data is always initialized with zeros. Returns the pointer that contains the array.

```
void iupArrayRemove ( Iarray * iarray,
                      int index,
                      int remove_count
                    )
```

Remove the number of elements from the array.

```
int iupArrayCount ( Iarray * iarray )
```

Returns the actual number of elements in the array.

Simple Draw API

[Utilities]

Collaboration diagram for Simple Draw API:

Functions

IdrawCanvas *	iupDrawCreateCanvas (Ihandle *ih)
void	iupDrawKillCanvas (IdrawCanvas *dc)
void	iupDrawFlush (IdrawCanvas *dc)
void	iupDrawUpdateSize (IdrawCanvas *dc)
void	iupDrawGetSize (IdrawCanvas *dc, int *w, int *h)
void	iupDrawParentBackground (IdrawCanvas *dc)
void	iupDrawLine (IdrawCanvas *dc, int x1, int y1, int x2, int y2, unsigned char r, unsigned char g, unsigned char b, int style)
void	iupDrawRectangle (IdrawCanvas *dc, int x1, int y1, int x2, int y2, unsigned char r, unsigned char g, unsigned char b, int style)
void	iupDrawArc (IdrawCanvas *dc, int x1, int y1, int x2, int y2, double a1, double a2, unsigned char r, unsigned char g, unsigned char b, int style)
void	iupDrawPolygon (IdrawCanvas *dc, int *points, int count, unsigned char r, unsigned char g, unsigned char b, int style)
void	iupDrawText (IdrawCanvas *dc, const char *text, int len, int x, int y, unsigned char r, unsigned char g, unsigned char b, const char *font)
void	iupDrawImage (IdrawCanvas *dc, const char *name, int make_inactive, int x, int y, int *img_w, int *img_h)
void	iupDrawSetClipRect (IdrawCanvas *dc, int x1, int y1, int x2, int y2)
void	iupDrawResetClip (IdrawCanvas *dc)
void	iupDrawSelectRect (IdrawCanvas *dc, int x, int y, int w, int h)
void	iupDrawFocusRect (IdrawCanvas *dc, int x, int y, int w, int h)

Detailed Description

See iup_draw.h

Function Documentation

IdrawCanvas* iupDrawCreateCanvas (Ihandle * *ih*)

Creates a draw canvas based on an IupCanvas. This will create an image for offscreen drawing.

void iupDrawKillCanvas (IdrawCanvas * *dc*)

Destroys the IdrawCanvas.

void iupDrawFlush (IdrawCanvas * *dc*)

Draws the ofscreen image on the screen.

void iupDrawUpdateSize (IdrawCanvas * *dc*)

Rebuild the offscreen image if the canvas size has changed. Automatically done in iupDrawCreateCanvas.

void iupDrawGetSize (IdrawCanvas * *dc*,
int * *w*,
int * *h*)

Returns the canvas size available for drawing.

void iupDrawParentBackground (IdrawCanvas * *dc*)

Draws the parent background.

void iupDrawLine (IdrawCanvas * *dc*,
int *x1*,
int *y1*,
int *x2*,
int *y2*,
unsigned char *r*,
unsigned char *g*,
unsigned char *b*,
int *style*)

Draws a line.

void iupDrawRectangle (IdrawCanvas * *dc*,
int *x1*,
int *y1*,
int *x2*,
int *y2*,
unsigned char *r*,
unsigned char *g*,
unsigned char *b*,
int *style*)

Draws a filled/hollow rectangle.

void iupDrawArc (IdrawCanvas * *dc*,
int *x1*,
int *y1*,
int *x2*,
int *y2*,
double *a1*,
double *a2*,
unsigned char *r*,
unsigned char *g*,
unsigned char *b*,
int *style*)

Draws a filled/hollow arc.

void iupDrawPolygon (IdrawCanvas * *dc*,
int * *points*,
int *count*,
unsigned char *r*,
unsigned char *g*,
unsigned char *b*,
int *style*)

Draws a filled/hollow polygon. points are arranged xyxyxy...

void iupDrawText	(	IdrawCanvas *	*dc*,	
		const char *	*text*,	
		int	*len*,	
		int	*x*,	
		int	*y*,	
		unsigned char	*r*,	
		unsigned char	*g*,	
		unsigned char	*b*,	
		const char *	*font*	
	)			

Draws a text. x,y is at left,top corner of the text.

void iupDrawImage	(	IdrawCanvas *	*dc*,	
		const char *	*name*,	
		int	*make_inactive*,	
		int	*x*,	
		int	*y*,	
		int *	*img_w*,	
		int *	*img_h*	
	)			

Draws an image. x,y is at left,top corner of the image. Returns the image size.

void iupDrawSetClipRect	(	IdrawCanvas *	*dc*,	
		int	*x1*,	
		int	*y1*,	
		int	*x2*,	
		int	*y2*	
	)			

Sets a rectangle clipping area.

void iupDrawResetClip	(	IdrawCanvas *	*dc*	)	

Removes clipping.

void iupDrawSelectRect	(	IdrawCanvas *	*dc*,	
		int	*x*,	
		int	*y*,	
		int	*w*,	
		int	*h*	
	)			

Draws a selection rectangle.

void iupDrawFocusRect	(	IdrawCanvas *	*dc*,	
		int	*x*,	
		int	*y*,	
		int	*w*,	
		int	*h*	
	)			

Draws a focus rectangle.

Generated on Wed Sep 16 2015 16:42:24 for SDK by doxygen *1.7.1*

Defines | Functions

String Utilities

[Utilities]

Collaboration diagram for String Utilities:

Defines

#define	iup_isdigit(_c) (_c>='0' && _c<='9')
#define	iup_toupper(_c) ((_c >= 'a' && _c <= 'z')? (_c - 'a') + 'A': _c)
#define	iup_tolower(_c) ((_c >= 'A' && _c <= 'Z')? (_c - 'A') + 'a': _c)
#define	IUP_FLOAT2STR "%.9f"
#define	IUP_DOUBLE2STR "%.18f"

Functions

int	iupStrEqual (const char *str1, const char *str2)
int	iupStrEqualNoCase (const char *str1, const char *str2)
int	iupStrEqualNoCaseNoSpace (const char *str1, const char *str2)
int	iupStrEqualPartial (const char *str1, const char *str2)
int	iupStrEqualNoCasePartial (const char *str1, const char *str2)
int	iupStrBoolean (const char *str)
int	iupStrFalse (const char *str)
int	iupStrLineCount (const char *str)
const char *	iupStrNextLine (const char *str, int *len)
const char *	iupStrNextValue (const char *str, int str_len, int *len, char sep)
int	iupStrCountChar (const char *str, char c)
char *	iupStrDup (const char *str)
char *	iupStrDupUntil (const char **str, char c)
void	iupStrCopyN (char *dst_str, int dst_max_size, const char *src_str)
char *	iupStrGetMemory (int size)
char *	iupStrGetLargeMem (int *size)
void	iupStrLower (char *dstr, const char *sstr)
void	iupStrUpper (char *dstr, const char *sstr)
int	iupStrHasSpace (const char *str)
int	iupStrIsAscii (const char *str)
char *	iupStrReturnStrf (const char *format,...)
char *	iupStrReturnStr (const char *str)
char *	iupStrReturnBoolean (int i)
char *	iupStrReturnChecked (int i)
char *	iupStrReturnInt (int i)
char *	iupStrReturnFloat (float f)
char *	iupStrReturnDouble (double d)
char *	iupStrReturnRGB (unsigned char r, unsigned char g, unsigned char b)
char *	iupStrReturnRGBA (unsigned char r, unsigned char g, unsigned char b, unsigned char a)
char *	iupStrReturnStrStr (const char *str1, const char *str2, char sep)
char *	iupStrReturnIntInt (int i1, int i2, char sep)
int	iupStrGetFormatPrecision (const char *format)
void	iupStrPrintfDoubleLocale (char *str, const char *format, double d, const char *decimal_symbol)
int	iupStrToRGB (const char *str, unsigned char *r, unsigned char *g, unsigned char *b)
int	iupStrToRGBA (const char *str, unsigned char *r, unsigned char *g, unsigned char *b, unsigned char *a)
int	iupStrToInt (const char *str, int *i)
int	iupStrToIntInt (const char *str, int *i1, int *i2, char sep)
int	iupStrToFloat (const char *str, float *f)
int	iupStrToDouble (const char *str, double *d)
int	iupStrToDoubleLocale (const char *str, double *d, const char *decimal_symbol)
int	iupStrToFloatFloat (const char *str, float *f1, float *f2, char sep)
int	iupStrToDoubleDouble (const char *str, double *f1, double *f2, char sep)
int	iupStrToStrStr (const char *str, char *str1, char *str2, char sep)
char *	iupStrFileGetExt (const char *file_name)
char *	iupStrFileGetTitle (const char *file_name)
char *	iupStrFileGetPath (const char *file_name)
char *	iupStrFileMakeFileName (const char *path, const char *title)
void	iupStrFileNameSplit (const char *filename, char *path, char *title)
int	iupStrReplace (char *str, char src, char dst)
void	iupStrToUnix (char *str)
void	iupStrToMac (char *str)
char *	iupStrToDos (const char *str)
char *	iupStrConvertToC (const char *str)
void	iupStrRemove (char *value, int start, int end, int dir, int utf8)
char *	iupStrInsert (const char *value, const char *insert_value, int start, int end, int utf8)
char *	iupStrProcessMnemonic (const char *str, char *c, int action)
int	iupStrFindMnemonic (const char *str)
int	iupStrCompare (const char *str1, const char *str2, int casesensitive, int utf8)
int	iupStrCompareEqual (const char *str1, const char *str2, int casesensitive, int utf8, int partial)
int	iupStrCompareFind (const char *str1, const char *str2, int casesensitive, int utf8)

Detailed Description

See iup_str.h

Define Documentation

```
#define iup_isdigit(_c)  (_c>='0' && _c<='9')
```

Checks if the character is a digit.

```
#define iup_toupper(_c)  ((_c >= 'a' && _c <= 'z')? (_c - 'a') + 'A': _c)
```

Converts a character into upper case.
It will work only for character codes <128.

```
#define iup_tolower(_c)  ((_c >= 'A' && _c <= 'Z')? (_c - 'A') + 'a': _c)
```

Converts a character into lower case.
It will work only for character codes <128.

```
#define IUP_FLOAT2STR  "%.9f"
```

maximum float precision

```
#define IUP_DOUBLE2STR  "%.18f"
```

maximum double precision

Function Documentation

```
int iupStrEqual(const char *str1,
                const char *str2)
```

Returns a non zero value if the two strings are equal. str1 or str2 can be NULL.

```
int iupStrEqualNoCase(const char *str1,
                      const char *str2)
```

Returns a non zero value if the two strings are equal but ignores case. str1 or str2 can be NULL. It will work only for character codes <128.

```
int iupStrEqualNoCaseNoSpace(const char *str1,
                             const char *str2)
```

Returns a non zero value if the two strings are equal but ignores case and spaces.
str1 or str2 can be NULL.
It will work only for character codes <128.

```
int iupStrEqualPartial(const char *str1,
                       const char *str2)
```

Returns a non zero value if the two strings are equal up to a number of characters defined by the strlen of the second string.
str1 or str2 can be NULL.

```
int iupStrEqualNoCasePartial(const char *str1,
                             const char *str2)
```

Returns a non zero value if the two strings are equal but ignores case up to a number of characters defined by the strlen of the second string.
str1 or str2 can be NULL.
It will work only for character codes <128.

```
int iupStrBoolean(const char *str)
```

Returns 1 if the string is "YES" or "ON".
Returns 0 otherwise (including NULL or empty).

```
int iupStrFalse(const char *str)
```

Returns 1 if the string is "NO" or "OFF".
Returns 0 otherwise (including NULL or empty).
To be used when value can be "False" or others different than "True".

```
int iupStrLineCount(const char *str)
```

Returns the number of lines in a string. It works for UNIX, DOS and MAC line ends.

```
const char* iupStrNextLine(const char *str,
                           int *len)
```

Returns a pointer to the next line and the size of the current line. It works for UNIX, DOS and MAC line ends. The size does not includes the line end. If str is NULL it will return NULL.

```
const char* iupStrNextValue(const char *str,
                            int str_len,
                            int *len,
                            char sep)
```

Returns a pointer to the next value and the size of the current value. The size does not includes the separator. If str is NULL it will return NULL.

```
int iupStrCountChar(const char *str,
```

		char	*c*	
	)			

Returns the number of repetitions of the character occours in the string.

char* iupStrDup	(	const char *	*str*	)

Returns a copy of the given string. If str is NULL it will return NULL.

char* iupStrDupUntil	(	const char **	*str*,	
		char	*c*	
	)			

Returns a new string containing a copy of the string up to the character. The string is then incremented to after the position of the character.

void iupStrCopyN	(	char *	*dst_str*,
		int	*dst_max_size*,
		const char *	*src_str*
	)		

Copy the string to the buffer, but limited to the max_size of the buffer. buffer is always properly ended.

char* iupStrGetMemory	(	int	*size*	)

Returns a buffer with the specified size+1.
The buffer is resused after 50 calls. It must NOT be freed. Use size=-1 to free all the internal buffers.

char* iupStrGetLargeMem	(	int *	*size*	)

Returns a very large buffer to be used in unknown size string construction. Use snprintf or vsnprintf with the given size.

void iupStrLower	(	char *	*dstr*,	
		const char *	*sstr*	
	)			

Converts a string into lower case. Can be used in-place.
It will work only for character codes <128.

void iupStrUpper	(	char *	*dstr*,	
		const char *	*sstr*	
	)			

Converts a string into upper case. Can be used in-place.
It will work only for character codes <128.

int iupStrHasSpace	(	const char *	*str*	)

Checks if the string has at least 1 space character.

int iupStrIsAscii	(	const char *	*str*	)

Checks if the string has only ASCII codes.

char* iupStrReturnStrf	(	const char *	*format*,	
			...	
	)			

Returns combined values in a formatted string using iupStrGetMemory. This is not supposed to be used for very large strings, just for combinations of numeric data or constant strings.

char* iupStrReturnStr	(	const char *	*str*	)

Returns a string value in a string using iupStrGetMemory.

char* iupStrReturnBoolean	(	int	*i*	)

Returns a boolean value (as YES or NO) in a string.

char* iupStrReturnChecked	(	int	*i*	)

Returns a checked value (as ON, OFF or NOTDEF (-1)) in a string.

char* iupStrReturnInt	(	int	*i*	)

Returns an int value in a string using iupStrGetMemory.

char* iupStrReturnFloat	(	float	*f*	)

Returns a float value in a string using iupStrGetMemory.

char* iupStrReturnDouble	(	double	*d*	)

Returns a double value in a string using iupStrGetMemory.

char* iupStrReturnRGB	(	unsigned char	*r*,	
		unsigned char	*g*,	
		unsigned char	*b*	
	)			

Returns a RGB value in a string using iupStrGetMemory.

char* iupStrReturnRGBA	(	unsigned char	*r*,
		unsigned char	*g*,
		unsigned char	*b*,
		unsigned char	*a*
	)		

Returns a RGBA value in a string using iupStrGetMemory.

char* iupStrReturnStrStr	(	const char *	*str1*,
		const char *	*str2*,
		char	*sep*
	)		

Returns two string values in a string using iupStrGetMemory.

char* iupStrReturnIntInt	(	int	*i1*,
		int	*i2*,
		char	*sep*
	)		

Returns two int values in a string using iupStrGetMemory.

int iupStrGetFormatPrecision	(	const char *	*format*	)

Returns the number of decimals in a format string for floating point output.

void iupStrPrintfDoubleLocale	(	char *	*str*,
		const char *	*format*,
		double	*d*,
		const char *	*decimal_symbol*
	)		

Prints a double in a string using the given decimal symbol.

int iupStrToRGB	(	const char *	*str*,
		unsigned char *	*r*,
		unsigned char *	*g*,
		unsigned char *	*b*
	)		

Extract RGB components from the string. Returns 0 or 1.

int iupStrToRGBA	(	const char *	*str*,
		unsigned char *	*r*,
		unsigned char *	*g*,
		unsigned char *	*b*,
		unsigned char *	*a*
	)		

Extract RGBA components from the string. Returns 0 or 1.

int iupStrToInt	(	const char *	*str*,
		int *	*i*
	)		

Converts the string to an int. The string must contains only the integer value. Returns a a non zero value if sucessfull.

int iupStrToIntInt	(	const char *	*str*,
		int *	*i1*,
		int *	*i2*,
		char	*sep*
	)		

Converts the string to two int. The string must contains two integer values in sequence, separated by the given character (usually 'x' or ':'). Returns the number of converted values. Values not extracted are not changed.

int iupStrToFloat	(	const char *	*str*,
		float *	*f*
	)		

Converts the string to a float. The string must contains only the real value. Returns a a non zero value if sucessfull.

int iupStrToDouble	(	const char *	*str*,
		double *	*d*
	)		

Converts the string to a double. The string must contains only the real value. Returns a a non zero value if sucessfull.

int iupStrToDoubleLocale	(	const char *	*str*,
		double *	*d*,
		const char *	*decimal_symbol*

	)			

Converts the string to a double using the given decimal symbol. The string must contains only the real value. Returns a a non zero value if sucessfull. Returns 2 if a locale was set.

int iupStrToFloatFloat	(	const char *	*str*,	
		float *	*f1*,	
		float *	*f2*,	
		char	*sep*	
	)			

Converts the string to two float. The string must contains two real values in sequence, separated by the given character (usually 'x' or ':'). Returns the number of converted values. Values not extracted are not changed. ATENTION: AVOID DEFINING THIS TYPE OF ATTRIBUTE VALUE.

int iupStrToDoubleDouble	(	const char *	*str*,	
		double *	*f1*,	
		double *	*f2*,	
		char	*sep*	
	)			

Converts the string to two double. The string must contains two real values in sequence, separated by the given character (usually 'x' or ':'). Returns the number of converted values. Values not extracted are not changed. ATENTION: AVOID DEFINING THIS TYPE OF ATTRIBUTE VALUE.

int iupStrToStrStr	(	const char *	*str*,	
		char *	*str1*,	
		char *	*str2*,	
		char	*sep*	
	)			

Extract two strings from the string. separated by the given character (usually 'x' or ':'). Returns the number of converted values. Values not extracted are set to empty strings.

char* iupStrFileGetExt	(	const char *	*file_name*	)

Returns the file extension of a file name. Supports UNIX and Windows directory separators.

char* iupStrFileGetTitle	(	const char *	*file_name*	)

Returns the file title of a file name. Supports UNIX and Windows directory separators.

char* iupStrFileGetPath	(	const char *	*file_name*	)

Returns the file path of a file name. Supports UNIX and Windows directory separators.

char* iupStrFileMakeFileName	(	const char *	*path*,	
		const char *	*title*	
	)			

Concat path and title addind '/' between if path does not have it.

void iupStrFileNameSplit	(	const char *	*filename*,	
		char *	*path*,	
		char *	*title*	
	)			

Split the filename in path and title using pre-alocated strings.

int iupStrReplace	(	char *	*str*,	
		char	*src*,	
		char	*dst*	
	)			

Replace a character in a string. Returns the number of occurrences.

void iupStrToUnix	(	char *	*str*	)

Convert line ends to UNIX format in-place (one
per line).

void iupStrToMac	(	char *	*str*	)

Convert line ends to MAC format in-place (one per line).

char* iupStrToDos	(	const char *	*str*	)

Convert line ends to DOS/Windows format (the sequence
per line). If returned pointer different the input, it must be freed.

char* iupStrConvertToC	(	const char *	*str*	)

Convert string to C format. Process
, and . If returned pointer different the input, it must be freed.

void iupStrRemove	(	char *	*value*,
		int	*start*,
		int	*end*,
		int	*dir*,
		int	*utf8*

	)			

Remove the interval from the string. Done in-place.

char* iupStrInsert	(	const char *	*value*,	
		const char *	*insert_value*,	
		int	*start*,	
		int	*end*,	
		int	*utf8*	
	)			

Remove the interval from the string and insert the new string at the start.

char* iupStrProcessMnemonic	(	const char *	*str*,	
		char *	*c*,	
		int	*action*	
	)			

Process the mnemonic in the string. If not found returns str. If found returns a new string. Action can be:

- 1: replace & by c
- -1: remove & and return in c
- 0: remove &

int iupStrFindMnemonic	(	const char *	*str*	)	

Returns the Mnemonic if found. Zero otherwise.

int iupStrCompare	(	const char *	*str1*,	
		const char *	*str2*,	
		int	*casesensitive*,	
		int	*utf8*	
	)			

Compare two strings using strcmp semantics, but using the "Alphanum Algorithm" (A1 A2 A11 A30 ...).
This means that numbers and text are sorted separately.
Also natural alphabetic order is used: 123...aAáÁ...bBcC...
Sorting and case insensitive will work only for Latin-1 characters, even when using utf8=1.

int iupStrCompareEqual	(	const char *	*str1*,	
		const char *	*str2*,	
		int	*casesensitive*,	
		int	*utf8*,	
		int	*partial*	
	)			

Returns a non zero value if the two strings are equal.
If partial=1 the compare up to a number of characters defined by the strlen of the second string.
Case insensitive will work only for Latin-1 characters, even when using utf8=1.

int iupStrCompareFind	(	const char *	*str1*,	
		const char *	*str2*,	
		int	*casesensitive*,	
		int	*utf8*	
	)			

Returns a non zero value if the second string is found inside the first string.
Uses iupStrCompareEqual.

- All
- Functions
- Typedefs
- Enumerations
- Enumerator
- Defines

- _
- i

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- _ -

- _IattribFlags : iup_class.h
- _IchildType : iup_class.h
- _InativeType : iup_class.h
- _Itable_IndexTypes : iup_table.h
- _Itable_Types : iup_table.h
